A hands-on Introduction to OpenCL

Tim Mattson

OpenCL

Acknowledgements: Alice Koniges of Berkeley Lab/NERSC and Simon McIntosh-Smith, James Price, and Tom Deakin of the University of Bristol

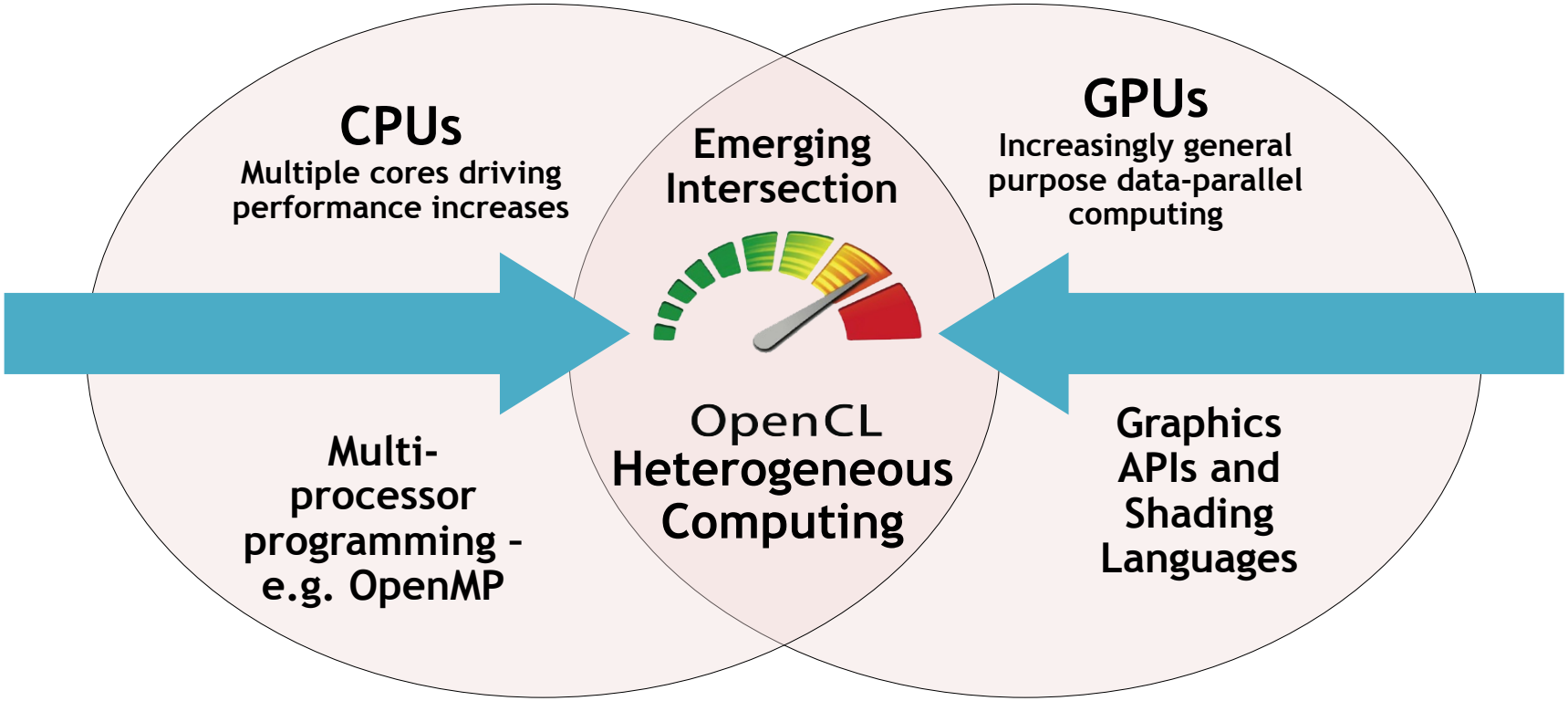
OpenCL Learning progression

Topic	Exercise	concepts
I. OCL intro		OpenCL overview, history and Core models.
II. Host programs	Vadd program	Understanding host programs
III. Kernel programs	Basic Jacobi solver	The OpenCL execution model and how it relates to kernel programs.
IV. Memory coalescence	Reorganizing the A matrix in the Jacobi solver program.	Memory layout effects on kernel performance
V. Divergent control flows	Divergent control flow in the Jacobi solver	Control flows and how they impact performance
VI. Occupancy	Work group size optimization for the Jacobi solver	Keeping all the resources busy
VII. Memory hierarchy in OpenCL	Demo: Matrix Multiplication	Working with private, local and global memory

Outline

- OpenCL: overview and core models
- Host programs
- Kernel programs
- Optimizing OpenCL kernels
  - Memory coelescence
  - Divergent control flows
  - Occupancy
  - Other Optimizations
- Working with the OpenCL Memory Hierarchy
- Resources supporting OpenCL

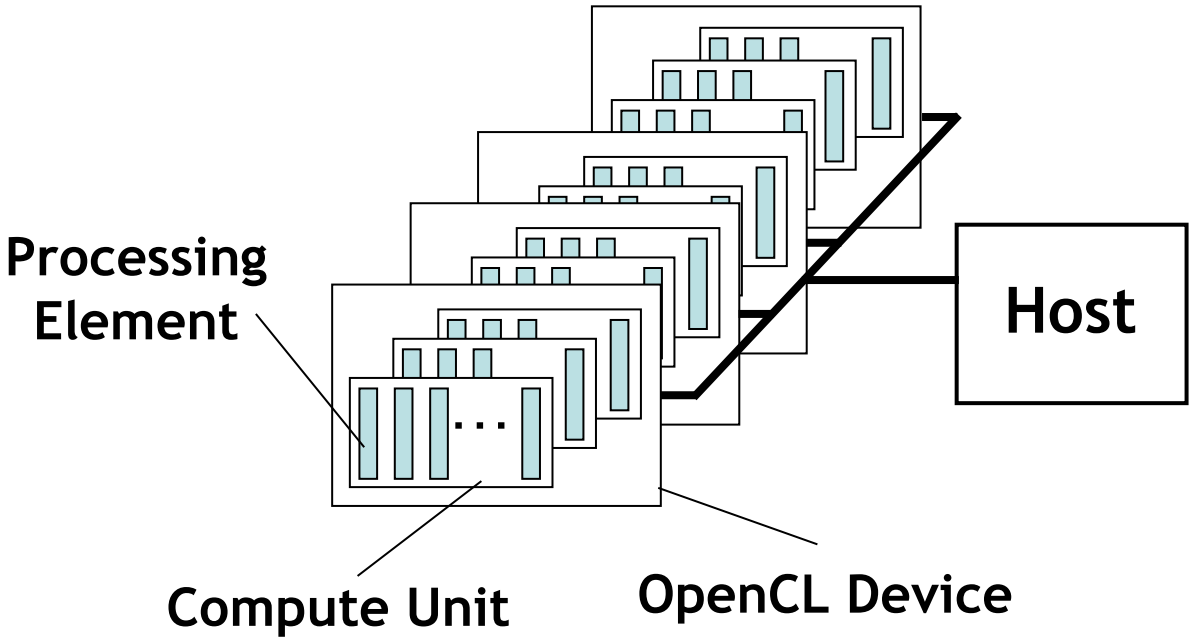
Industry Standards for Programming Heterogeneous Platforms



OpenCL – Open Computing Language

Open, royalty-free standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors

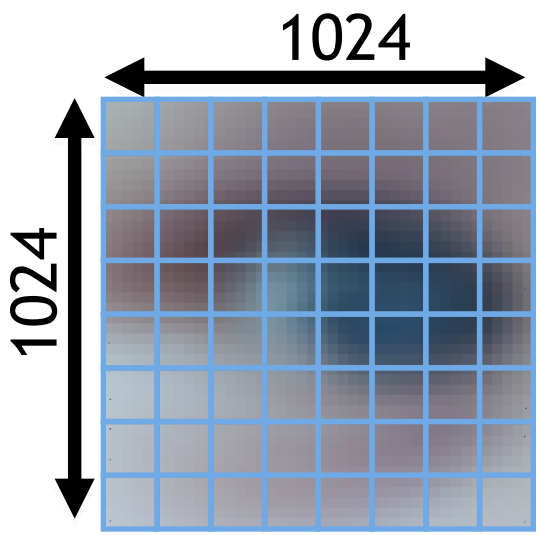
OpenCL Platform Model



- One *Host* and one or more *OpenCL Devices*
  - Each OpenCL Device is composed of one or more *Compute Units*
    - Each Compute Unit is divided into one or more *Processing Elements*
- Memory divided into *host memory* and *device memory*

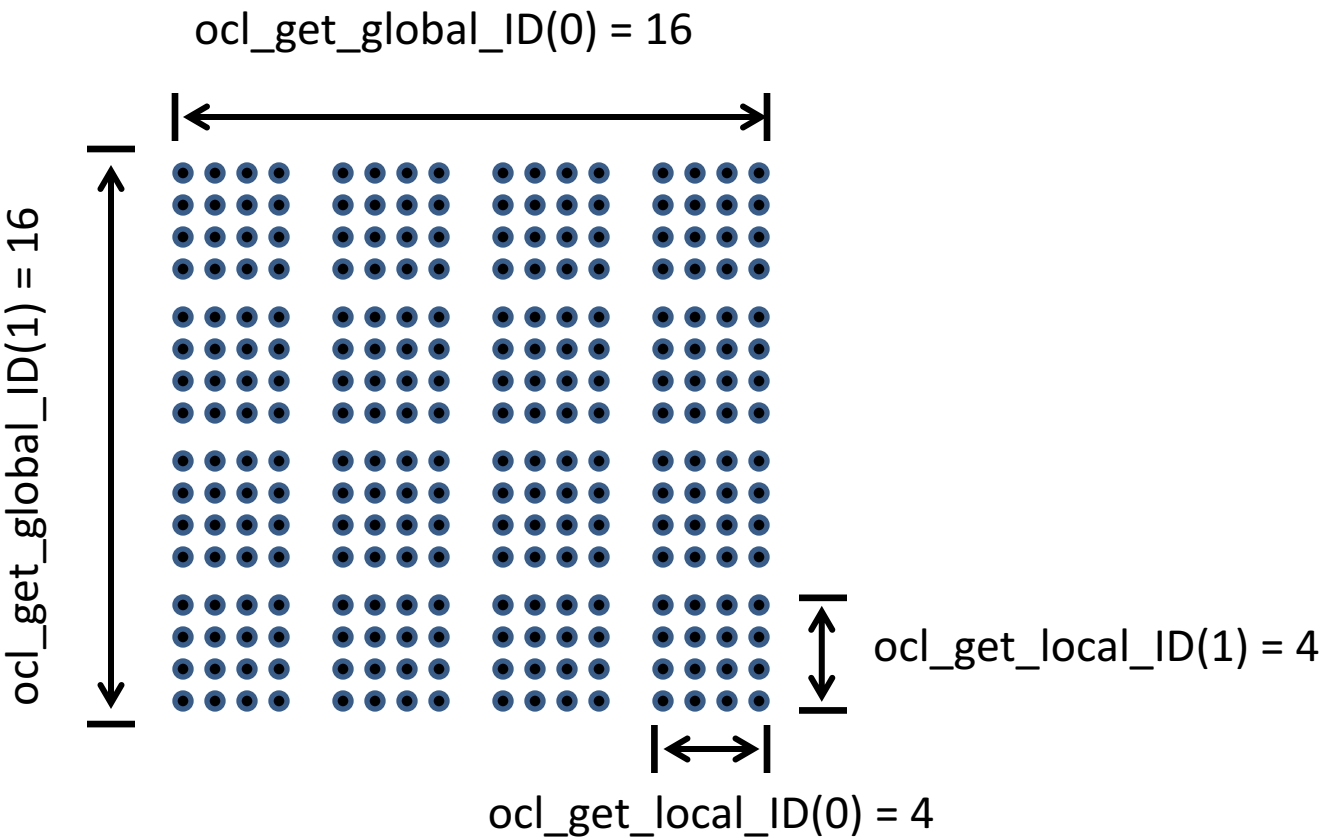
An N-dimensional domain of work-items

- **Global** Dimensions:
  - 1024x1024 (whole problem space)
- **Local** Dimensions:
  - 128x128 (**work-group**, executes together)



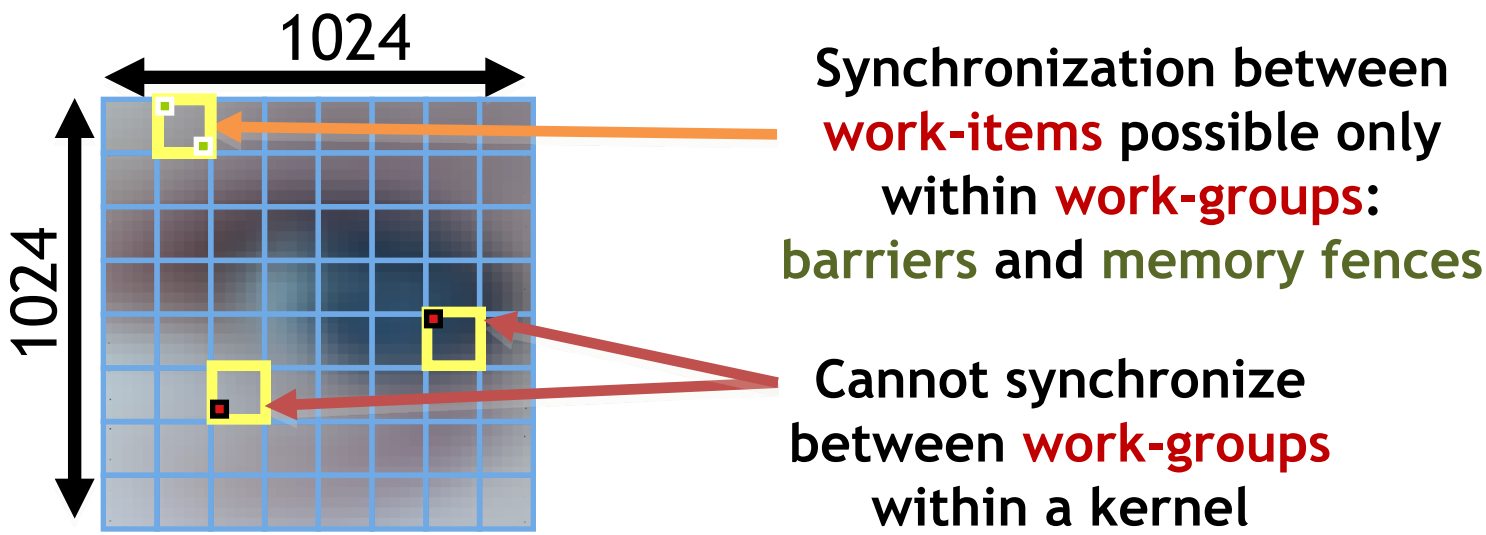
- Choose the dimensions (1, 2, or 3) that are "best" for your algorithm

Index-space/work-items/work-groups



An N-dimensional domain of work-items

- **Global** Dimensions:
  - 1024x1024 (whole problem space)
- **Local** Dimensions:
  - 128x128 (**work-group**, executes together)

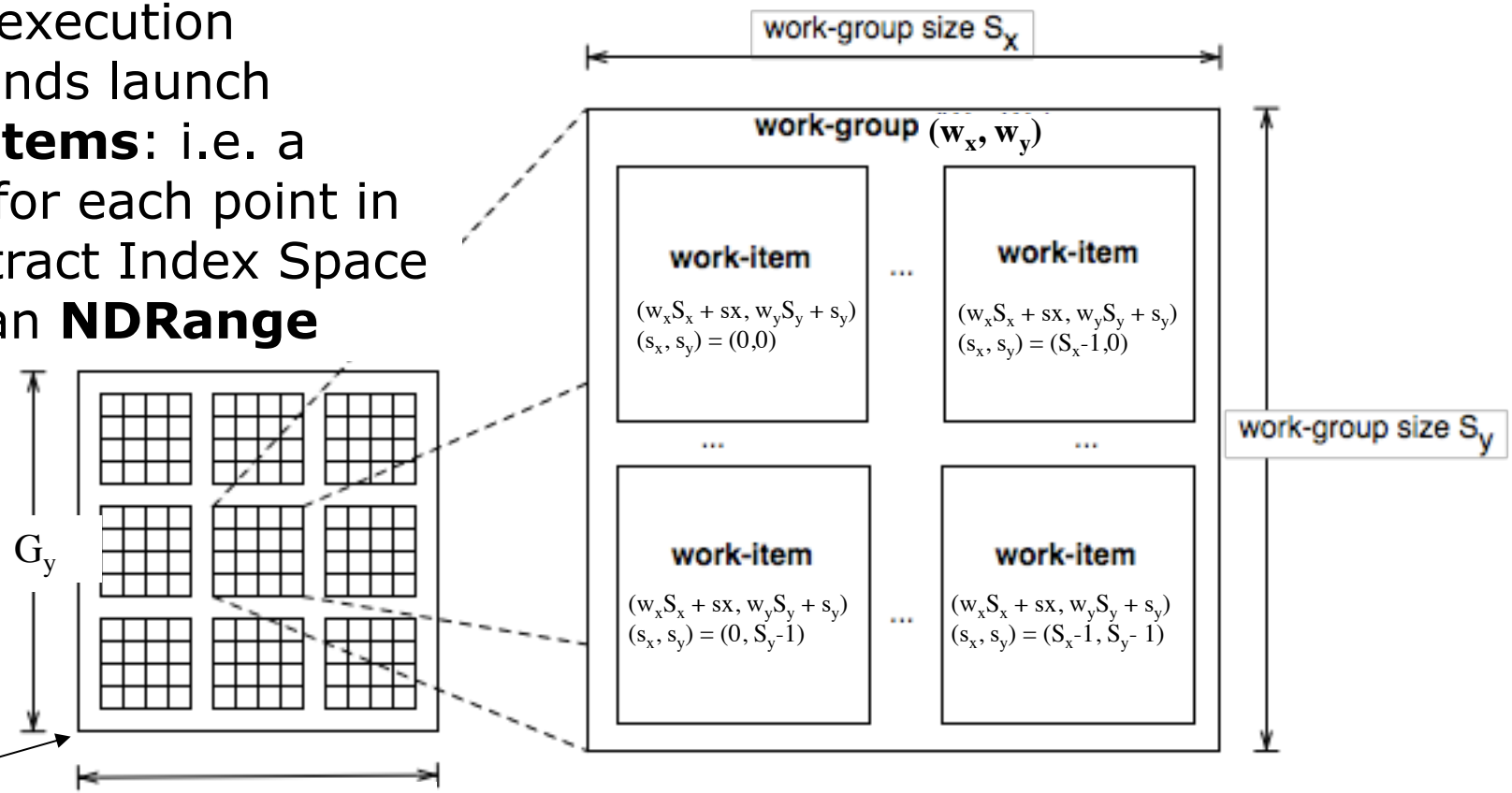


- Choose the dimensions (1, 2, or 3) that are "best" for your algorithm

Execution Model

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue

Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**

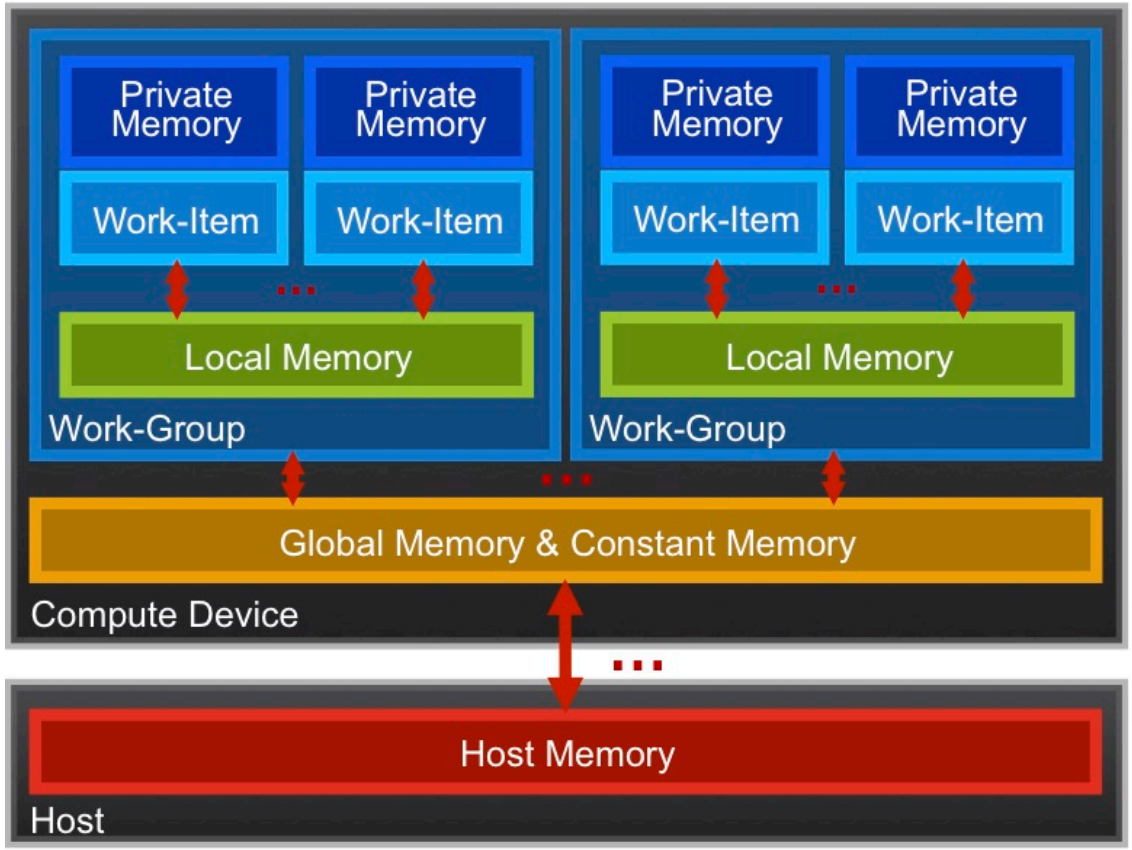


A (G_y by G_x) index space

Work items execute together as a **work-group**.

OpenCL Memory model

- ***Private Memory***
  - Per work-item
- ***Local Memory***
  - Shared within a work-group
- ***Global Memory /Constant Memory***
  - Visible to all work-groups
- ***Host memory***
  - On the CPU



Memory management is **explicit**:
You are responsible for moving data from
host → global → local *and* back

The BIG idea behind OpenCL

- Replace loops with functions (a kernel) executing at each point in a problem domain.
  - E.g., process a 1024 x 1024 image with one kernel invocation per pixel or 1024 x 1024 = 1,048,576 kernel executions

Traditional loops

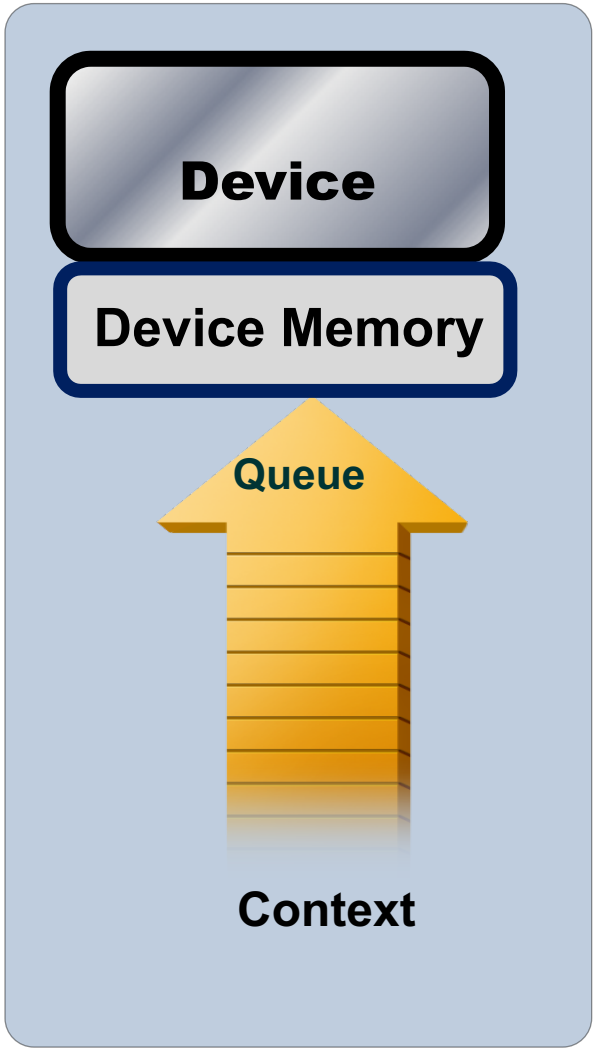
```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

Context and Command-Queues

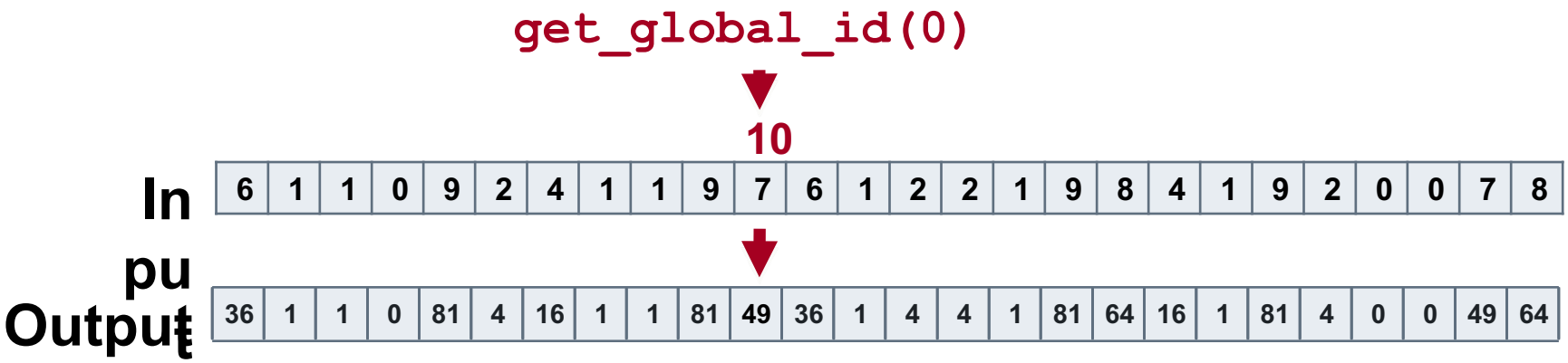
- Context:
  - The environment within which kernels execute and in which synchronization and memory management is defined.
- The context includes:
  - One or more devices
  - Device memory
  - One or more command-queues
- All commands for a device (kernel execution, synchronization, and memory operations) are submitted through a command-queue.
- Each Command-queue points to a single device within a context.



Execution model (kernels)

- OpenCL execution model … define a problem domain and execute an instance of a kernel for each point in the domain

```
kernel void square(
      global float* input,
      global float* output)
{
  int i = get_global_id(0);
  output[i] = input[i] * input[i];
}
```



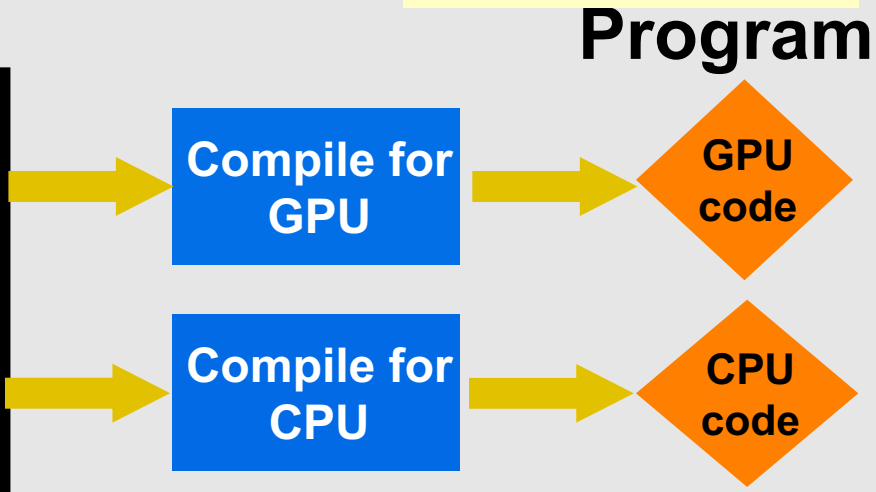
Building Program objects

- The program object encapsulates:
  - A context
  - The program source/binary
  - List of target devices and build options
- The Build process … to create a program object
  - clCreateProgramWithSource()
  - clCreateProgramWithBinary()

OpenCL uses runtime compilation … because in general you don't know the details of the device when you ship the program

Kernel Code

```
kernel void
horizontal_reflect(read_only image2d_t src,
                   write_only image2d_t dst)
{
  int x = get_global_id(0);  // x-coord
  int y = get_global_id(1);  // y-coord
  int width = get_image_width(src);
  float4 src_val = read_imagef(src, sampler,
                       (int2)(width-1-x, y));
  write_imagef(dst, (int2)(x, y), src_val);
}
```



Example: vector addition

- The “hello world” program of data parallel programming is a program to add two vectors

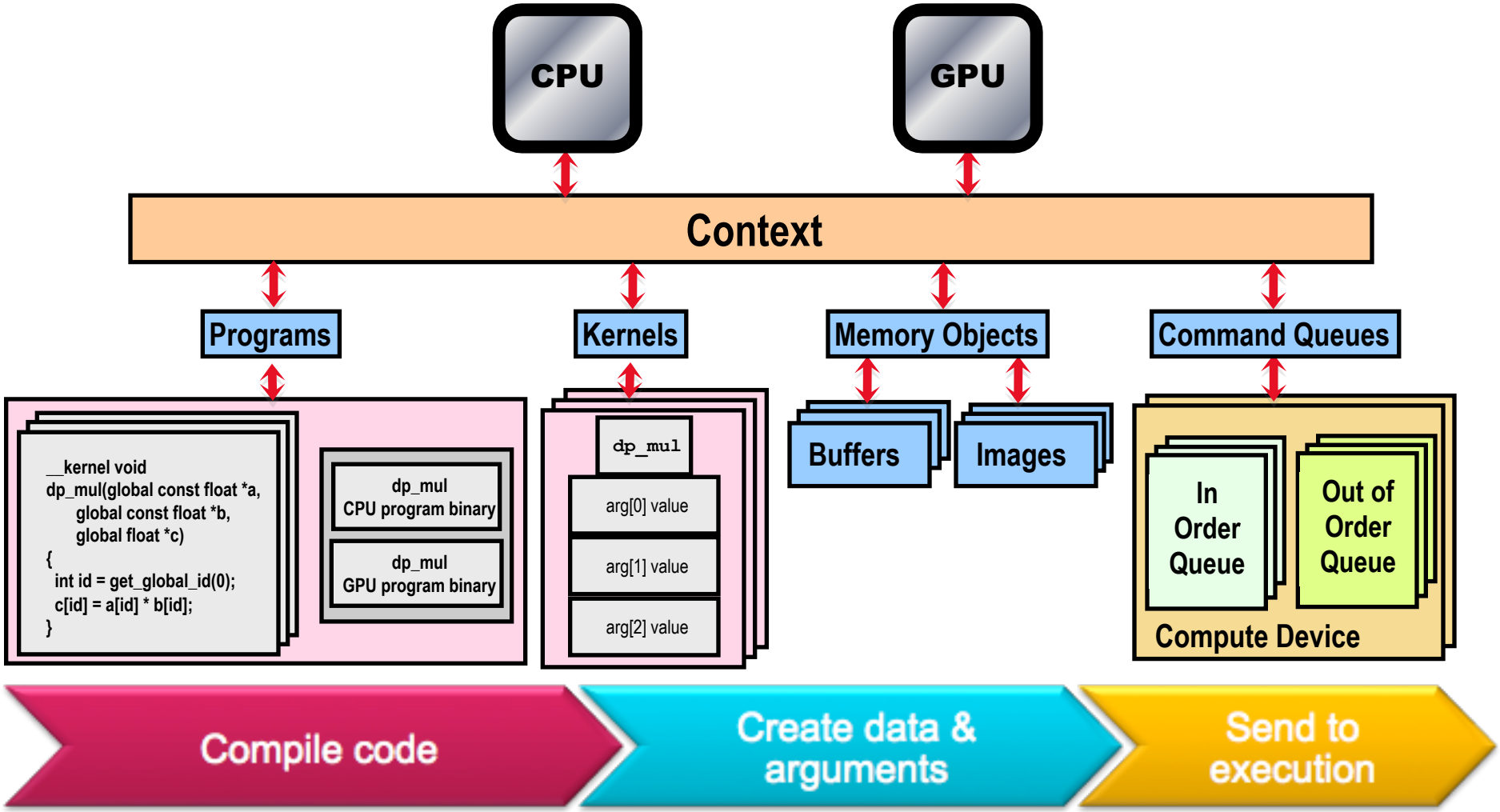
$$C[i] = A[i] + B[i] \quad \text{for } i=1 \text{ to } N$$

- For the OpenCL solution, there are two parts
  - Kernel code
  - Host code

Vector Addition - Kernel

```
__kernel void vadd (__global const float *a,
                    __global const float *b,
                    __global       float *c)
 {
     int gid = get_global_id(0);
     c[gid]  = a[gid] + b[gid];
}
```

The basic platform and runtime APIs in OpenCL (using C)



Outline

- OpenCL: overview and core models
- Host programs
- Kernel programs
- Optimizing OpenCL kernels
  - Memory coelescence
  - Divergent control flows
  - Occupancy
  - Other Optimizations
- Working with the OpenCL Memory Hierarchy
- Resources supporting OpenCL

Vector Addition - Host

- The host program … the code that runs on the host to:
  - Setup the environment for the OpenCL program
  - Create and manage kernels
- 5 simple steps in a basic Host program

1. Define the platform … platform = devices+context+queues
2. Create and Build the program (dynamic library for kernels)
3. Setup memory objects
4. Define kernel (attach arguments to kernel function)
5. Submit commands … transfer memory objects and execute kernels

1. Define the platform

- **Grab the first available Platform:**

```
err = clGetPlatformIDs(1, &firstPlatformId, &numPlatforms);
```

- **Use the first CPU device the platform provides:**

```
err = clGetDeviceIDs(firstPlatformId, CL_DEVICE_TYPE_CPU, 1,
                                                  &device_id, NULL);
```

- **Create a simple context with a single device:**

```
context = clCreateContext(firstPlatformId, 1, &device_id, NULL,
                                                  NULL, &err);
```

- **Create a simple command queue to feed our compute device:**

```
commands = clCreateCommandQueue(context, device_id, 0, &err);
```

2. Create and Build the program

- **Define source code for the kernel-program as a string literal (great for toy programs) or read from a file (common in real apps).**
- **Build the program object:**

```
program = clCreateProgramWithSource(context, 1,
        (const char **) & KernelSource, NULL, &err);
```

- **Compile the program to create a "dynamic library" from which specific kernels can be pulled:**

```
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

- **Fetch and print error messages (if(err != CL_SUCCESS)):**

```
size_t len;        char buffer[2048];
clGetProgramBuildInfo(program, device_id,
                      CL_PROGRAM_BUILD_LOG, sizeof(buffer),

buffer, &len);
printf("%s\n", buffer);
```

3. Setup Memory Objects

- For vector addition, 3 memory objects … one for each input vector (A and B) and one for the output vector (C).
- Create input vectors and assign values on the host:

```
float       a_data[LENGTH], b_data[LENGTH], c_res [LENGTH];
for(i = 0; i < count; i++){
        a_data[i] = rand() / (float)RAND_MAX;
        b_data[i] = rand() / (float)RAND_MAX;
}
```

- **Define OpenCL memory objects:**

```
a_in   = clCreateBuffer(context,  CL_MEM_READ_ONLY,
                                  sizeof(float) * count, NULL, NULL);
b_in   = clCreateBuffer(context,  CL_MEM_READ_ONLY,
                                  sizeof(float) * count, NULL, NULL);
c_out  = clCreateBuffer(context,  CL_MEM_WRITE_ONLY,
                                  sizeof(float) * count, NULL, NULL);
```

4. Define the kernel

- Create kernel object from the kernel function “vadd”:

```
kernel = clCreateKernel(program, "vadd", &err);
```

- **Attach arguments to the kernel function “vadd” to memory objects:**

```
err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_in);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_in);
err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_out);
err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &count);
```

5. Submit commands

- **Write Buffers from host into global memory (as non-blocking operations)**

```
err = clEnqueueWriteBuffer( commands, a_in, CL_FALSE, 0,
                            sizeof(float) * count, a_data, 0, NULL, NULL );
err = clEnqueueWriteBuffer( commands, b_in, CL_FALSE, 0,
                            sizeof(float) * count, b_data, 0, NULL, NULL );
```

- **Enqueue the kernel for execution (note: in-order queue so this is OK)**

```
err = clEnqueueNDRangeKernel( commands, kernel, 1, NULL,
                              &global, &local, 0, NULL, NULL );
```

- **Read back the result (as a blocking operation). Use the fact that we have an in-order queue which assures the previous commands are done before the read begins.**

```
err = clEnqueueReadBuffer( commands, c_out, CL_TRUE, 0,
                           sizeof(float) * count, c_res, 0, NULL, NULL );
```

Vector Addition - Host Program

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
   CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
                                        NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
   devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
   0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA,
                                          NULL);}
memobjs[1] = clCreateBuffer(context,CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcB,
                                          NULL);
memobjs[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                           sizeof(cl_float)*n, NULL,
                                          NULL);
// create the program
program = clCreateProgramWithSource(context, 1,
   &program_source, NULL, NULL);
```

```
// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL,
                                            NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err  = clSetKernelArg(kernel, 0, (void *) &memobjs[0],
                                  sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                  sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                   sizeof(cl_mem));
// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2],
   CL_TRUE, 0, n*sizeof(cl_float), dst, 0, NULL, NULL);
```

Vector Addition - Host Program

```
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
   CL_DEVICE_TYPE_GPU, NULL, NULL, NULL);

// get the list of GPU devices associated with context
c
d
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb,
   devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0],
   0, NULL);
```

Define platform and queues

```
// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY |
   CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, srcA,
   NULL);}
memo                                        ONLY |
   (                                        cB,
   b
memobjs[2] = clCreateBuffer(context,CL_MEM_WRITE_ONLY,
                              sizeof(cl_float)*n, NULL,
   NULL);
// create the program
```

Define memory objects

```
program = clCreateProgramWithSource(context, 1,
   &pr
```

Create the program

```
// build
err = cl                                  L, NULL,
   NULL);
```

Build the program

```
// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the arg values
err                                          js[0],
                                             ;
err |= clSetKernelArg(kernel, 1, (void *)&memobjs[1],
                                 sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *)&memobjs[2],
                                  sizeof(cl_mem));
```

Create and setup kernel

```
// set work-item dimensions
global_work_size[0] = n;

// execu
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1,
   NULL, global_work_size, NULL, 0, NULL, NULL);
```

Execute the kernel

```
// rea
err =                                        TRUE,
   0,
```

Read results on the host

It's complicated, but most of this is "boilerplate" and not as bad as it looks.

Exercise 1: Running the Vadd kernel

- Goal:
  - To inspect and verify that you can build and run an OpenCL kernel
- Procedure:
  - Use the vadd.c program and makefile we provide.

```
make vadd
```

  - Run he program. It will run a simple kernel to add two vectors together.

```
./vadd
```

  - Look at the host code and identify the API calls discussed in these slides.
- Expected output:
  - A message verifying that the vector addition completed successfully

For our OpenCL exercises, use our GPU server: urania.pd.infn.it

Outline

- OpenCL: overview and core models
- Host programs
- Kernel programs
- Optimizing OpenCL kernels
  - Memory coelescence
  - Divergent control flows
  - Occupancy
  - Other Optimizations
- Working with the OpenCL Memory Hierarchy
- Resources supporting OpenCL

Kernel programming

- Kernel programming is where all the action is at in OpenCL
- Writing simple OpenCL kernels is quite easy, so we'll cover that quickly
- Optimizing OpenCL kernels to run really fast is much harder, so that's where we're going to spend most of the time

OpenCL C kernel language

- Derived from **ISO C99**
  - A few ***restrictions***: no recursion, function pointers, functions in C99 standard headers ...
  - Preprocessing directives defined by C99 are supported (#include etc.)

- Built-in data types
  - Scalar and vector data types, pointers
  - Data-type conversion functions:
    - convert_type<_sat><_roundingmode>
  - Image types: image2d_t, image3d_t and sampler_t

OpenCL C Language Highlights

- Function qualifiers
  - **__kernel** qualifier declares a function as a kernel
    - I.e. makes it visible to host code so it can be enqueued
  - Kernels can call other kernel-side functions

- Address space qualifiers
  - **__global, __local, __constant, __private**
  - Pointer kernel arguments must be declared with an address space qualifier

- Work-item functions
  - **uint get_work_dim() … number of dimensions in use (1,2, or 3)**
  - **size_t get_global_id(uint n) … global work-item ID in dim "n"**
  - **size_t get_local_id(uint n) … work-item ID in dim "n" inside work-group**
  - **size_t get_group_id(uint n) … ID of work-group in dim "n"**
  - **size_t get_global_size(uint n) … num of work-items in dim "n"**
  - **size_t get_local_size(uint n) … num of work-items in work group in dim "n"**

- Synchronization functions
  - **Barriers** - all work-items within a work-group must execute the barrier function before any work-item can continue
  - **Memory fences** - provides ordering between memory operations

OpenCL C Language Restrictions

- Pointers to functions are ***not*** allowed
- Pointers to pointers allowed ***within*** a kernel, but not as an argument to a kernel invocation
- Bit-fields are not supported
- Variable length arrays and structures are not supported
- Recursion is not supported (yet!)
- Double types are optional in OpenCL v1.2, but the key word is reserved

  (note: most implementations support double)

Memory Consistency

- OpenCL uses a **relaxed consistency** memory model; i.e.
  - The state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.
- Within a work-item:
  - Memory has load/store consistency to the work-item's private view of memory, i.e. it sees its own reads and writes correctly
- Within a work-group:
  - Local memory is consistent between work-items at a barrier.
- Global memory is consistent within a work-group at a barrier, **but *not* guaranteed across different work-groups!!**
  - This is a common source of bugs!
- Consistency of memory shared between commands (e.g. kernel invocations) is enforced by synchronization (barriers, events, in-order queue)

Work-Item Synchronization

Ensure correct order of memory operations to local or global memory (with flushes or queuing a memory fence)

- Within a work-group

  `void barrier()`

  – Takes optional flags

    CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE

  – A work-item that encounters a **barrier**() will wait until ALL work-items in its work-group reach the **barrier**()

  – **Corollary**: If a **barrier**() is inside a branch, then the branch must be taken by either:

    - **ALL** work-items in the work-group, OR
    - **NO** work-item in the work-group

- Across work-groups

  – No guarantees as to where and when a particular work-group will be executed relative to another work-group

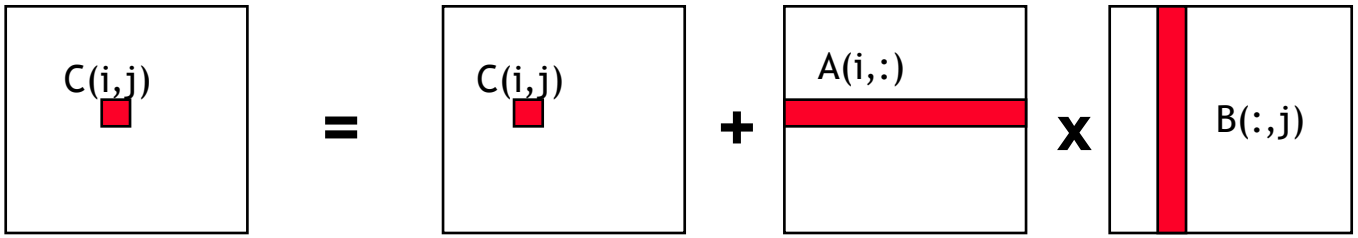
  – **Cannot exchange data, or have barrier-like synchronization between two different work-groups! (Critical issue!)**

  – **Only solution: finish the kernel and start another**

Matrix multiplication: sequential code

We calculate C=AB, dimA = (N x N), dimB=(N x N), dimC=(N x N)

```
void mat_mul(int Order, float *A, float *B, float *C)
{
    int i, j, k;

    for (i = 0; i < Order; i++) {
      for (j = 0; j < Order; j++) {
        for (k = 0; k < Order; k++) {
           // C(i, j) = sum(over k) A(i,k) * B(k,j)
           C[i*Order+j] += A[i*Order+k] * B[k*Order+j];
        }
      }
    }
}
```



Dot product of a row of A and a column of B for each element of C

Matrix multiplication performance

- Serial C code on CPU (single core).

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A

Device is Intel® Xeon® CPU, E5649 @ 2.53GHz using the gcc compiler.

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Matrix multiplication: sequential code

```
void mat_mul(int Order, float *A, float *B, float *C)
{
    int i, j, k;

    for (i = 0; i < Order; i++) {
      for (j = 0; j < Order; j++) {
        for (k = 0; k < Order; k++) {
          // C(i, j) = sum(over k) A(i,k) * B(k,j)
          C[i*Order+j] += A[i*Order+k] * B[k*Order+j];
        }
      }
    }
}
```

Matrix multiplication: Kernel code (1/2)

```
__kernel void mat_mul(
            const int Order, __global float *A,
           __global float *B, __global float *C)
{
    int i, j, k;

    for (i = 0; i < Order; i++) {
      for (j = 0; j < Order; j++) {
        for (k = 0; k < Order; k++) {
          // C(i, j) = sum(over k) A(i,k) * B(k,j)
          C[i*Order+j] += A[i*Order+k] * B[k*Order+j];
        }
      }
    }
}
```

Mark as a kernel function and specify memory qualifiers

Matrix multiplication: Kernel code (2/2)

```
__kernel void mat_mul(
            const int Order, __global float *A,
           __global float *B, __global float *C)
{
    int i, j, k;

    i = get_global_id(0);
    j = get_global_id(1);

        for (k = 0; k < Order; k++) {
          // C(i, j) = sum(over k) A(i,k) * B(k,j)
          C[i*Order+j] += A[i*Order+k] * B[k*Order+j];
        }

}
```

Remove outer loops and set work-item co-ordinates

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Exercise: Jacobi Solver Program

- Goal:
  - To write a non-trivial OpenCL kernel
- Procedure:
  - Consider the serial program jac_solv.c. Look at the program, run it, and understand what's its doing.
  - The program Jac_solv_ocl_basic.c is a C host program to run an OpenCL kernel for the jacobi solver.
  - A "skeleton" of the kernel program is in the file jac_ocl_basic.cl.
  - Inside the file jac_ocl_basic.cl, write the body of the kernel program.
- Expected output:
  - A message verifying that the program ran correctly .

Jacobi solver kernel code (1/2)

```
#define TYPE float

kernel void jacobi(
        const unsigned Ndim,
        global TYPE * A,  global TYPE * b,
        global TYPE * xold,  global TYPE * xnew)

{
    size_t i = get_global_id(0);

    xnew[i] = (TYPE) 0.0;
    for (int j = 0; j < Ndim; j++)  {
        if (i != j)
            xnew[i] += A[i*Ndim + j] * xold[j];
    }
    xnew[i] = (b[i] - xnew[i]) / A[i*Ndim + i];}
```

Jacobi solver kernel code (2/2)

```
kernel void convergence(
        global TYPE * xold,  global TYPE * xnew,
         local TYPE * conv_loc,  global TYPE * conv  )
{
      size_t i = get_global_id(0);
      TYPE tmp;
       tmp = xnew[i] - xold[i];
       conv_loc[get_local_id(0)] = tmp * tmp;
       barrier(CLK_LOCAL_MEM_FENCE);

       for (int offset = get_local_size(0) / 2; offset > 0; offset /= 2)  {
          if (get_local_id(0) < offset)    {
              conv_loc[get_local_id(0)] += conv_loc[get_local_id(0) + offset];
          }
          barrier(CLK_LOCAL_MEM_FENCE);
       }
       if (get_local_id(0) == 0)  {    conv[get_group_id(0)] = conv_loc[0];  }
}
```

A kernel enqueued on the host to compute convergence. This implements a reduction with the last stage of the reduction occuring on the host.

Jacobi Solver Results

- Serial code running on the Intel® Xeon® CPU and icc took 83 seconds.
- With OpenMP for multithreading
  - 25.3 seconds with 32 threads (hyperthreading enabled)
  - 19.0 seconds with 16 threads (hyperthreading disabled)
- Running the OpenMP version natively on the Intel® Xeon® Phi Processor took 4.8 seconds.

Different versions of the Jacobi Solver with OpenCL. Runtimes in seconds				
TYPE = double NDIM = 4096	**Nvidia K40 GPU**	**AMD 290X GPU**	**Intel Xeon PHI processor**	**Intel Xeon processor**
Basic	35.0	198.2	245.2	23.6

Dual-socket Intel® Xeon® CPU E5-2687W (16 cores total, hyper-threading enabled) and the Intel® icc compiler.

Outline

- OpenCL: overview and core models
- Host programs
- Kernel programs
- Optimizing OpenCL kernels
  - Memory coelescence
  - Divergent control flows
  - Occupancy
  - Other Optimizations
- Working with the OpenCL Memory Hierarchy
- Resources supporting OpenCL

Coalesced Access

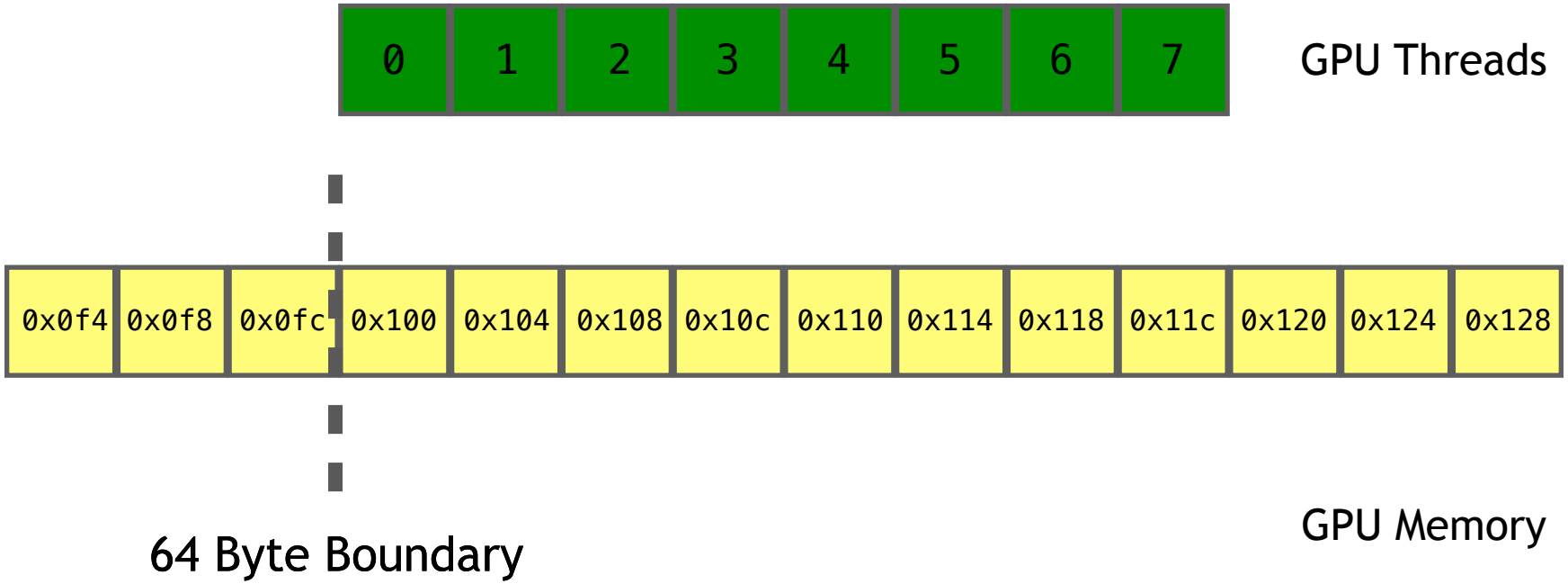
- ***Coalesced memory accesses*** are key for high performance code
  - Goal: if thread *i* accesses memory location *n* then thread *i+1* accesses memory location *n+1*
- In principle, it's very simple, but frequently requires transposing/transforming data on the host before sending it to the GPU

Coalescence

- Coalesced memory accesses are key for high bandwidth
- Simply, it means, if thread *i* accesses memory location *n* then thread *i+1* accesses memory location *n+1*
- In practice, it's not quite as strict…

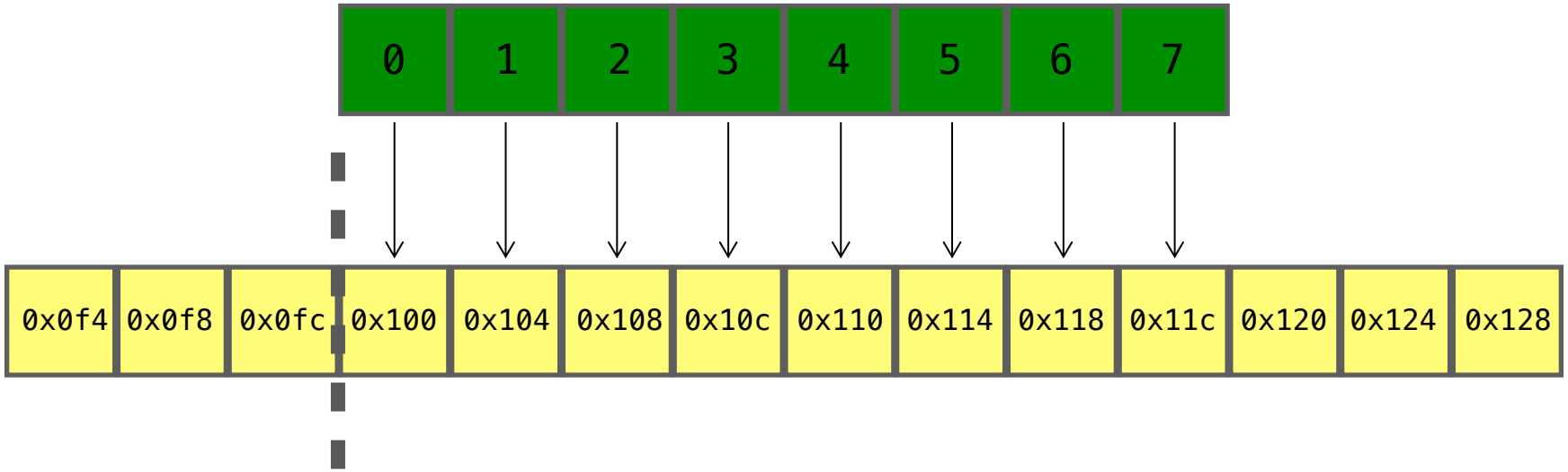
```
__kernel func( __global float *memA,
               __global float *memB)
{
      int g_id = get_global_id(0);

      // ideal
      float val1 = memA[g_id];

      // still pretty good
      const int c = 3;
      float val2 = memA[g_id + c];

      // stride size is not so good
      float val3 = memA[c*g_id];

      const int loc =
        some_strange_func(g_id);

      // terrible!
      float val4 = memA[loc];
}
```

Memory access patterns



Memory access patterns

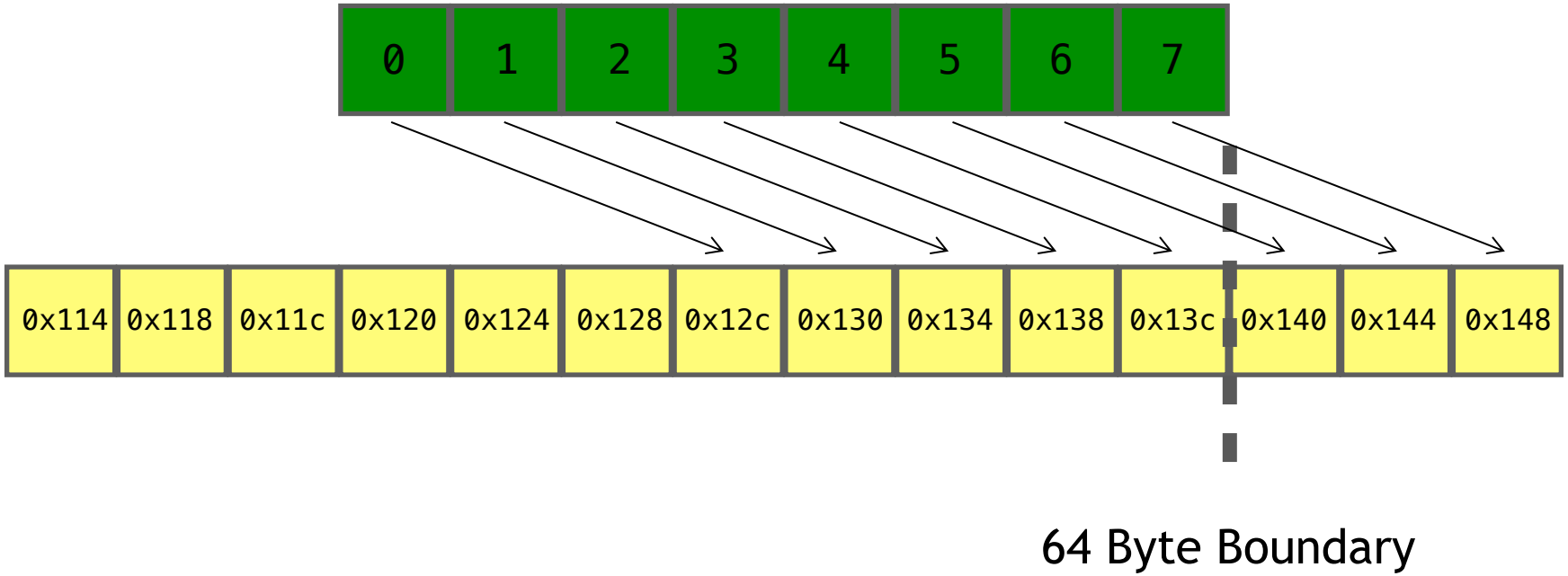
```
float val1 = memA[g_id];
```



64 Byte Boundary

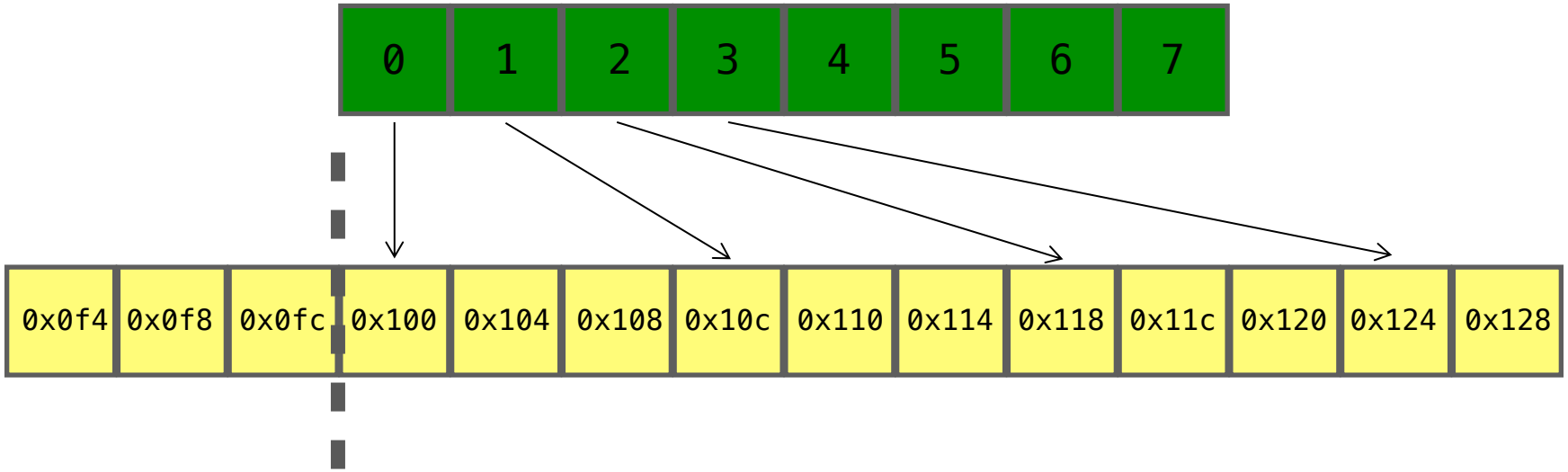
Memory access patterns

```
const int c = 3;
float val2 = memA[g_id + c];
```



Memory access patterns

```
float val3 = memA[3*g_id];
```



64 Byte Boundary

Strided access results in multiple memory transactions (and kills throughput)

Coalesced Access

- ***Coalesced memory accesses*** are key for high performance code
- In principle, it's very simple, but frequently requires transposing/transforming data on the host before sending it to the GPU
- Sometimes this is an issue of AoS vs. SoA

Memory layout is critical to performance

- "Structure of Arrays vs. Array of Structures" problem
- Array of Structures (AoS) more natural to code

```
struct Point{ float x, y, z, a; };
Point *Points;
```

x	y	z	a	…	x	y	z	a	…	x	y	z	a	…	x	y	z	a	…

- Structure of Arrays (SoA) suits memory coalescence in vector units

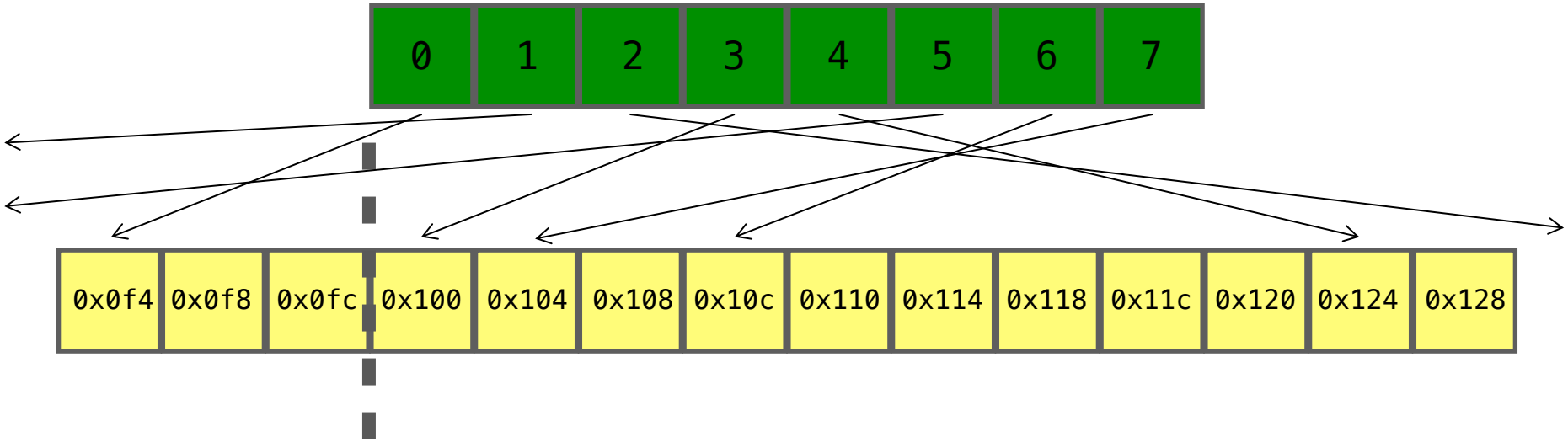
```
struct { float *x, *y, *z, *a; } Points;
```

x	x	x	x	…	y	y	y	y	…	z	z	z	z	…	a	a	a	a	…

Adjacent work-items/vector-lanes like to access adjacent memory locations

Memory access patterns

```
const int loc =  some_strange_func(g_id);

float val4 = memA[loc];
```



64 Byte Boundary

Exercise

- Inspect the memory access patterns in your Jacobi solver kernel.
- Is there a memory alignment problem? If so, fix it.
- If you want to generate the transpose of the A matrix (a column major order), we provide a function inside mm_utils.c that you can call inside the host code to do this.

```
void init_colmaj_diag_dom_near_identity_matrix(int Ndim,  TYPE *A);
```

Jacobi solver kernel code

```
#define TYPE float
#if (TYPE == double)
    #pragma OPENCL EXTENSION cl_khr_fp64 : enable
#endif

Kernel void jacobi(
        const unsigned Ndim,
        global TYPE * A,  global TYPE * b,
        global TYPE * xold,  global TYPE * xnew)

{
    size_t I = get_global_id(0);

    xnew[i] = (TYPE) 0.0;
    for (int j = 0; j < Ndim; j++)  {
        if (I != j)
            xnew[i] += A[j*Ndim + i] * xold[j];
    }
    xnew[i] = (b[i] – xnew[i]) / A[i*Ndim + i];}
```

Original code (row-major A) was:

xnew[i] += A[i*Ndim + j] * xold[j];

Adjacent work-items process offset locations into A

Switch to a column-major A matrix so adjacent work-items process adjacent locations in A as you go through the loop over j

Jacobi Solver Results

- Serial code running on the Intel® Xeon® CPU and icc took 83 seconds.
- With OpenMP for multithreading
  - 25.3 seconds with 32 threads (hyperthreading enabled)
  - 19.0 seconds with 16 threads (hyperthreading disabled)
- Running the OpenMP version natively on the Intel® Xeon® Phi Processor took 4.8 seconds.

Different versions of the Jacobi Solver with OpenCL. Runtimes in seconds				
TYPE = double NDIM = 4096	**Nvidia K40 GPU**	**AMD 290X GPU**	**Intel Xeon PHI processor**	**Intel Xeon processor**
Basic	35.0	198.2	245.2	23.6
Colmaj	14.1	15.3	35.8	71.5

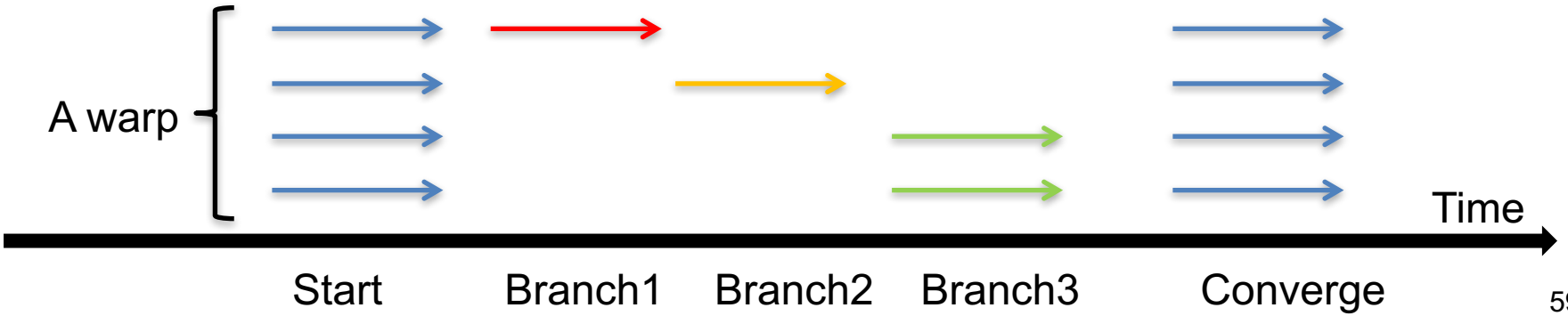
Dual-socket Intel® Xeon® CPU E5-2687W (16 cores total, hyper-threading enabled) and the Intel® icc compiler.

Outline

- OpenCL: overview and core models
- Host programs
- Kernel programs
- Optimizing OpenCL kernels
  - Memory coelescence
  - Divergent control flows
  - Occupancy
  - Other Optimizations
- Working with the OpenCL Memory Hierarchy
- Resources supporting OpenCL

Single Instruction Multiple Data

- Individual threads of a warp start together at the same program address
- Each thread has its own instruction address counter and register state
  - Each thread is free to branch and execute independently
  - Provide the MIMD abstraction
- Branch behavior
  - Each branch will be executed serially
  - Threads not following the current branch will be disabled



Branching

- GPUs tend not to support speculative execution, which means that branch instructions have high latency
- This latency can be hidden by switching to alternative work-items/work-groups, but avoiding branches where possible is still a good idea to improve performance
- When different work-items executing within the same SIMD ALU array take different paths through conditional control flow, we have *divergent branches (vs. uniform branches)*
- These are even worse: work-items will stall while waiting for the others to complete
- We can use predication, selection and masking to convert conditional control flow into straight line code and significantly improve the performance of code that has lots of conditional branches

Exercise

- Eliminate the branch in your Jacobi solver kernel.
- We don’t need any host change so use the same host program as last time:
  - Jac_solv_ocl_colmaj.c

Branching

Conditional execution

```
// Only evaluate expression
// if condition is met
if (a > b)
{
  acc += (a - b*c);
}
```

Selection and masking

```
// Always evaluate expression
// and mask result
temp = (a - b*c);
mask = (a > b ? 1.f : 0.f);
acc += (mask * temp);
```

Jacobi solver kernel code

```
#define TYPE double
#if (TYPE == double)
    #pragma OPENCL EXTENSION cl_khr_fp64 : enable
#endif

kernel void jacobi(
        const unsigned Ndim,
        global TYPE * A,  global TYPE * b,
        global TYPE * xold,  global TYPE * xnew)

{
    size_t i = get_global_id(0);

    xnew[i] = (TYPE) 0.0;
    for (int j = 0; j < Ndim; j++)  {

        xnew[i] += A[j*Ndim + i] * xold[j] * (TYPE)(i != j);
    }
    xnew[i] = (b[i] - xnew[i]) / A[i*Ndim + i];}
```

Jacobi Solver Results

- Serial code running on the Intel® Xeon® CPU and icc took 83 seconds.
- With OpenMP for multithreading
  - 25.3 seconds with 32 threads (hyperthreading enabled)
  - 19.0 seconds with 16 threads (hyperthreading disabled)
- Running the OpenMP version natively on the Intel® Xeon® Phi Processor took 4.8 seconds.

Different versions of the Jacobi Solver with OpenCL. Runtimes in seconds				
TYPE = double NDIM = 4096	**Nvidia K40 GPU**	**AMD 290X GPU**	**Intel Xeon PHI processor**	**Intel Xeon processor**
Basic	35.0	198.2	245.2	23.6
Colmaj	14.1	15.3	35.8	71.5
No Branch	13.3	15.6	16.6	38.8

Note: optimizations in the table are cumulative

Dual-socket Intel® Xeon® CPU E5-2687W (16 cores total, hyper-threading enabled) and the Intel® icc compiler.

Outline

- OpenCL: overview and core models
- Host programs
- Kernel programs
- Optimizing OpenCL kernels
  - Memory coelescence
  - Divergent control flows
  - Occupancy
  - Other Optimizations
- Working with the OpenCL Memory Hierarchy
- Resources supporting OpenCL

Keep the processing elements (PE) busy

- **Occupancy**: a measure of the fraction of time during a computation when the PE's are busy. Goal is to keep this number high (well over 50%).
- Pay attention to the number of work-items and work-group sizes
  - Rule of thumb: On a modern GPU you want at least 4 work-items per PE in a Compute Unit
  - More work-items are better, but diminishing returns, and there is an upper limit
    - Each work item consumes PE finite resources (registers etc)

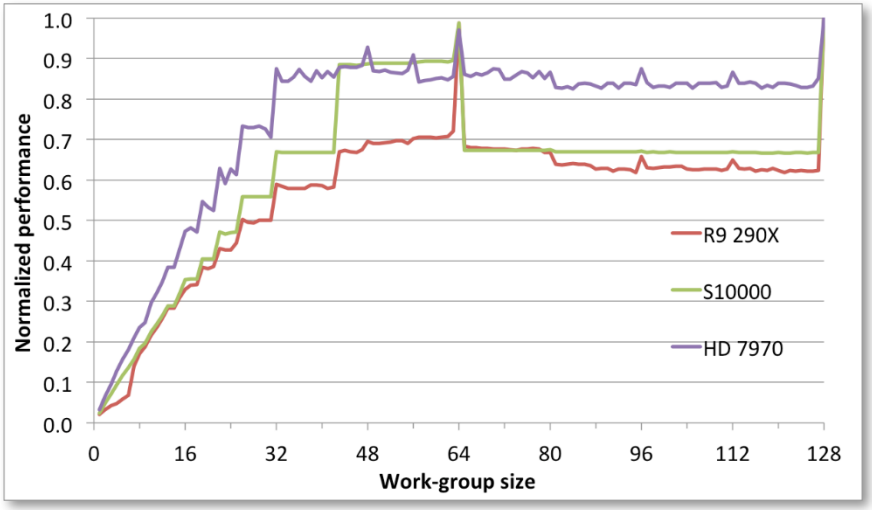
Occupancy

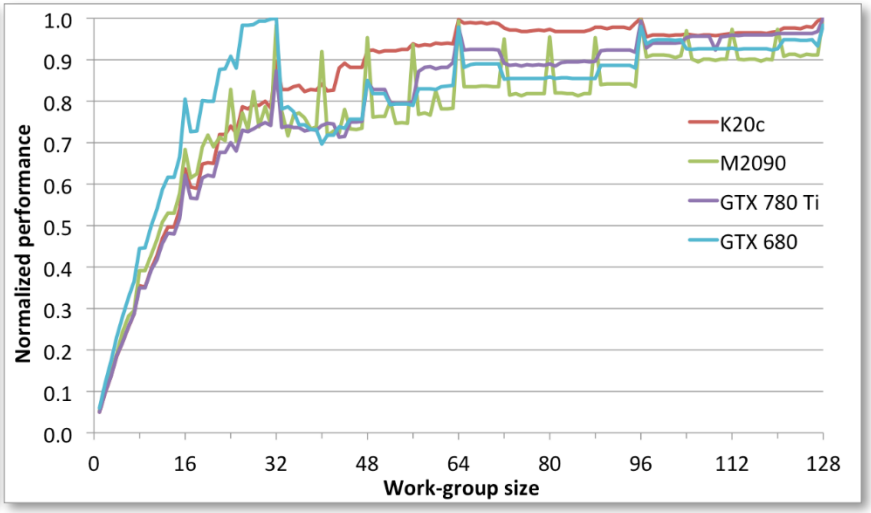
- Number of work-groups per compute unit (CU) depends on registers and local memory size per work-group
- E.g. NVIDIA's K40 has 128 words of memory per processor element (PE), i.e. 128 registers per core; and 48KB of local memory per CU
- But, multiple work-items (threads) will be scheduled on a single PE (similar to hyperthreading)
- In fact, global memory latency is so high that multiple work-items per PE are a *requirement* for achieving a good proportion of peak performance!

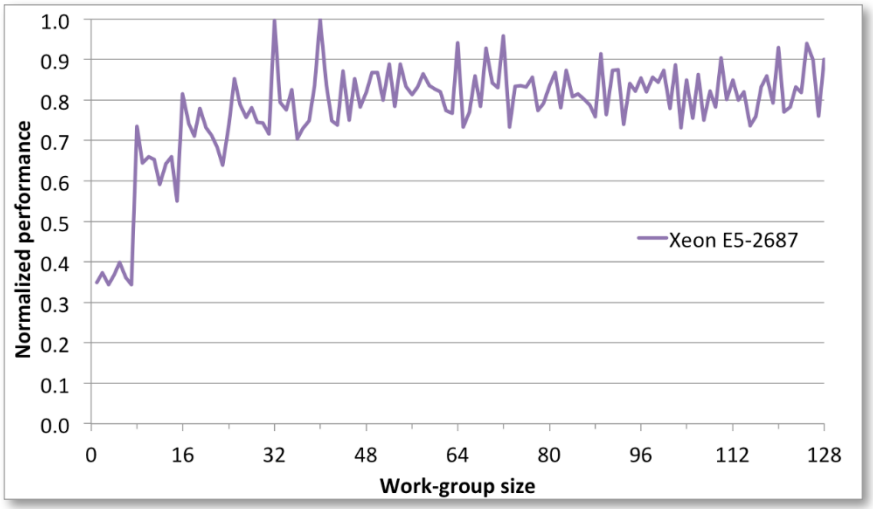
Work-group sizes

- Work-group sizes being a power of 2 helps on most architectures. At a minimum use multiples of:
  - 8 for Intel® AVX CPUs
  - 16 for Intel® Xeon Phi™ processors
  - 32 for Nvidia® GPUs
  - 64 for AMD®
  - May be different on different hardware
- On most systems aim to run lots of work-groups. For example, on Xeon Phi, multiples of the number of threads available (e.g. 240 on a 5110P) is optimal, but as many as possible is good (1000+)

Effect of work-group sizes







Exercise

- Experiment with different work group sizes. Use host program

  jac_solv_colmaj_nobr_wg.c
- You do not need to change the kernel program ... Use your kernel program from the last exercise.
- Run the host program with the flag –h to see the command line options. One of them (--wg) will vary the workgroup size.

Jacobi Solver Results

- Serial code running on the Intel® Xeon® CPU and icc took 83 seconds.
- With OpenMP for multithreading
  - 25.3 seconds with 32 threads (hyperthreading enabled)
  - 19.0 seconds with 16 threads (hyperthreading disabled)
- Running the OpenMP version natively on the Intel® Xeon® Phi Processor took 4.8 seconds.

Different versions of the Jacobi Solver with OpenCL. Runtimes in seconds				
TYPE = double NDIM = 4096	**Nvidia K40 GPU**	**AMD 290X GPU**	**Intel Xeon PHI processor**	**Intel Xeon processor**
Basic	35.0	198.2	245.2	23.6
Colmaj	14.1	15.3	35.8	71.5
No Branch	13.3	15.6	16.6	38.8
Opt WG size	13.2	15.1	15.0	32.1

Note: optimizations in the table are cumulative
Dual-socket Intel® Xeon® CPU E5-2687W (16 cores total, hyper-threading enabled) and the Intel® icc compiler.

Outline

- OpenCL: overview and core models
- Host programs
- Kernel programs
- Optimizing OpenCL kernels
  - Memory coelescence
  - Divergent control flows
  - Occupancy
  - Other Optimizations
- Working with the OpenCL Memory Hierarchy
- Resources supporting OpenCL

Constant Memory

- Constant memory can be considered a store for data that never changes
- Setting and updating constants in memory uses the same interface as global memory, with enqueueRead/enqueueWrite commands
- The difference is how it is declared in the kernel
- Some devices may have dedicated on-chip caches or data-paths for constant memory
- Devices are guaranteed to support constant memory allocations of at least 64kB
- Can also declare OpenCL program scope constant data, but this has to be initialized at OpenCL program compile time

```
kernel void
calc_something
(
  global float *a,
  global float *b,
  global float *c,

  //constant memory is
  //set on the host
  constant float *params
)
{
  //code here
}
```

Compiler Options

- OpenCL compilers accept a number of flags that affect how kernels are compiled:

```
-cl-opt-disable
-cl-single-precision-constant
-cl-denorms-are-zero
-cl-fp32-correctly-rounded-divide-sqrt
-cl-mad-enable
-cl-no-signed-zeros
-cl-unsafe-math-optimizations
-cl-finite-math-only
-cl-fast-relaxed-math
```

implies

Other compilation hints

- Can use an attribute to inform the compiler of the work-group size that you intend to launch kernels with:

```
__attribute__((reqd_work_group_size(x, y, z)))
```

- As with C/C++, use the `const`/`restrict` keywords for kernel arguments where appropriate to make sure the compiler can optimise memory accesses

Exercise

- Experiment with different optimizations to get the best runtime you can.

Jacobi Solver Results

- Serial code running on the Intel® Xeon® CPU and icc took 83 seconds.
- With OpenMP for multithreading
  - 25.3 seconds with 32 threads (hyperthreading enabled)
  - 19.0 seconds with 16 threads (hyperthreading disabled)
- Running the OpenMP version natively on the Intel® Xeon® Phi Processor took 4.8 seconds.

Different versions of the Jacobi Solver with OpenCL. Runtimes in seconds				
TYPE = double NDIM = 4096	**Nvidia K40 GPU**	**AMD 290X GPU**	**Intel Xeon PHI processor**	**Intel Xeon processor**
Basic	35.0	198.2	245.2	23.6
Colmaj	14.1	15.3	35.8	71.5
No Branch	13.3	15.6	16.6	38.8
Opt WG size	13.2	15.1	15.0	32.1
Unroll by 4	6.2	6.7	13.3	32.1

Note: optimizations in the table are cumulative

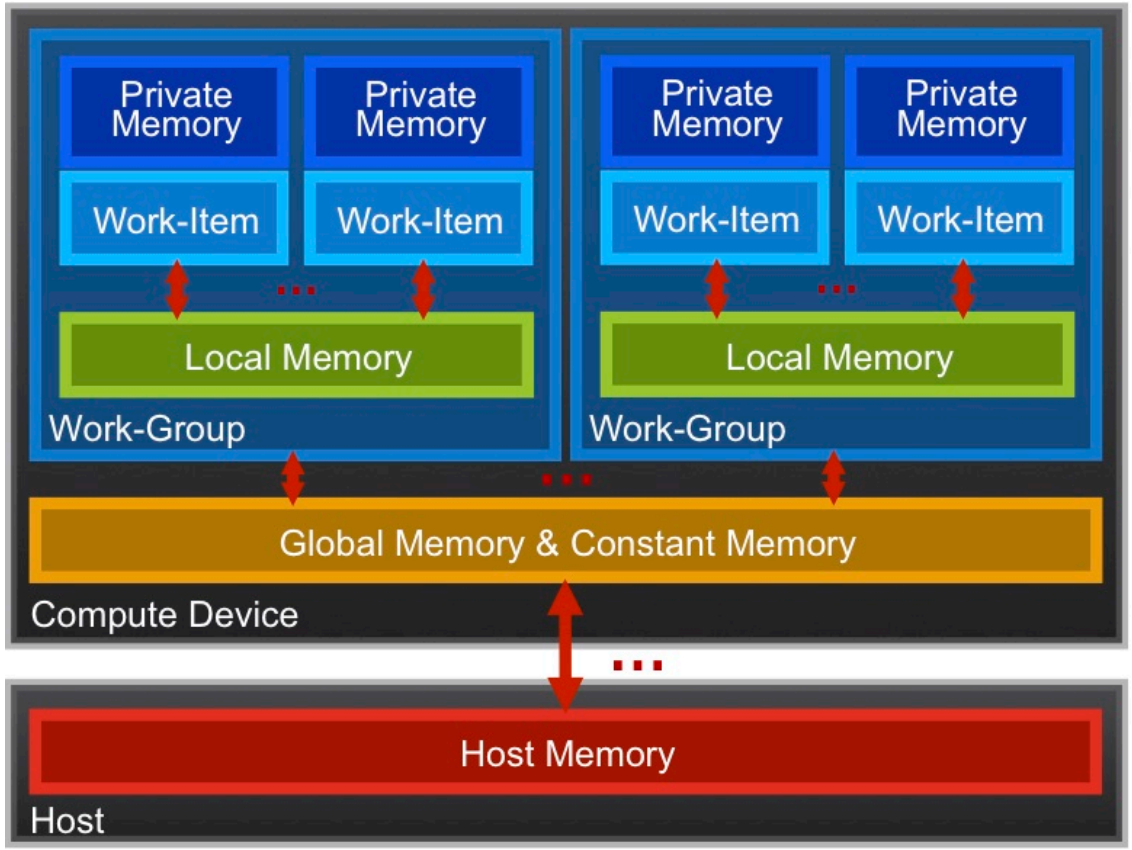
Dual-socket Intel® Xeon® CPU E5-2687W (16 cores total, hyper-threading enabled) and the Intel® icc compiler.

Outline

- OpenCL: overview and core models
- Host programs
- Kernel programs
- Optimizing OpenCL kernels
  - Memory coelescence
  - Divergent control flows
  - Occupancy
  - Other Optimizations
- Working with the OpenCL Memory Hierarchy
- Resources supporting OpenCL

OpenCL Memory model

- Private Memory
  - Per work-item
- Local Memory
  - Shared within a work-group
- Global/Constant Memory
  - Visible to all work-groups
- Host memory
  - On the CPU



Memory management is **explicit**:
You are responsible for moving data from
host → global → local *and* back

The Memory Hierarchy

Bandwidths

Private memory
O(2-3) words/cycle/WI

Local memory
O(10) words/cycle/WG

Global memory
O(100-200) GBytes/s

Host memory
O(1-100) GBytes/s

Sizes

Private memory
O(10) words/WI

Local memory
O(1-10) KBytes/WG

Global memory
O(1-10) GBytes

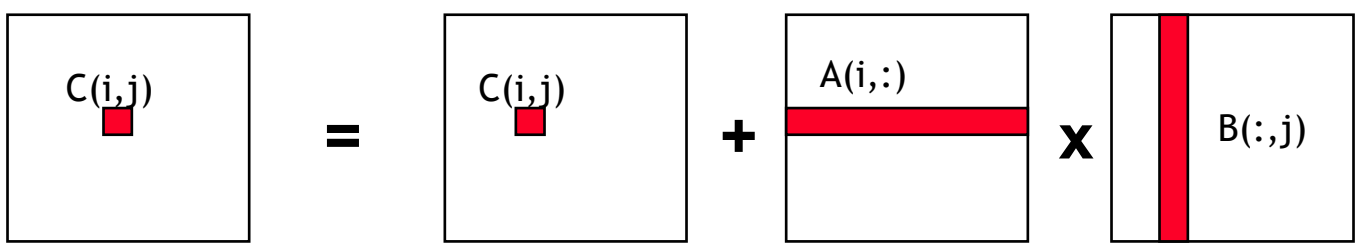
Host memory
O(1-100) GBytes

Managing the memory hierarchy is one of ***the*** most important things to get right to achieve good performance

*Size and performance numbers are approximate and for a high-end discrete GPU, circa 2011

Optimizing matrix multiplication

- MM cost determined by FLOPS and memory movement:
  - $2*n^3 = O(n^3)$ FLOPS
  - Operates on $3*n^2 = O(n^2)$ numbers
- To optimize matrix multiplication, we must ensure that for every memory access we execute as many FLOPS as possible.
- Outer product algorithms are faster, but for pedagogical reasons, let's stick to the simple dot-product algorithm.

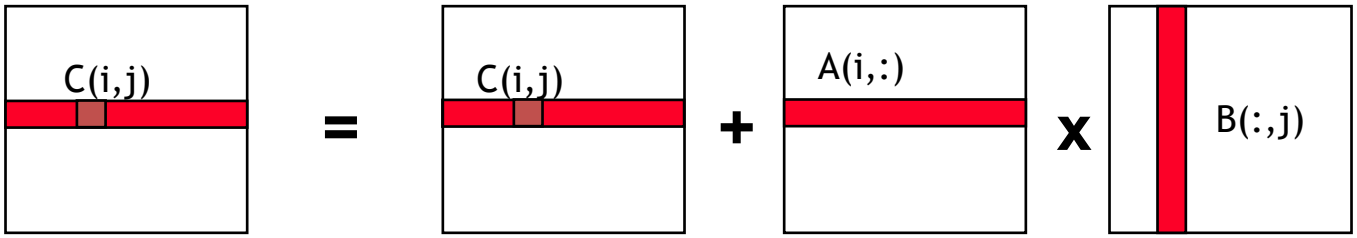


Dot product of a row of A and a column of B for each element of C

- We will work with work-item/work-group sizes and the memory model to optimize matrix multiplication

Optimizing matrix multiplication

- There may be significant overhead to manage work-items and work-groups.
- So let's have each work-item compute a full row of C

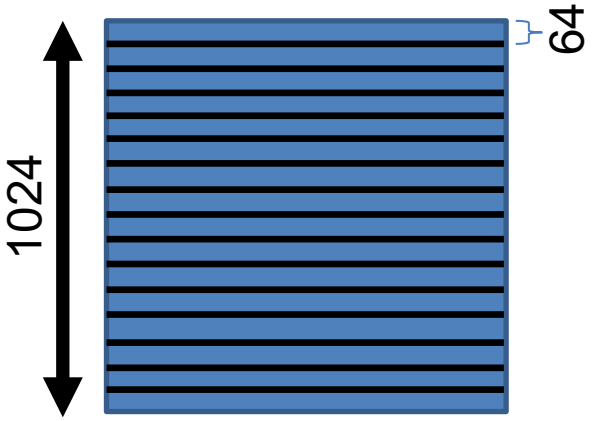


Dot product of a row of A and a column of B for each element of C

- And with an eye towards future optimizations, let's collect work-items into work-groups with 64 work-items per work-group

An N-dimension domain of work-items

- Global Dimensions: 1024 (1D)
  Whole problem space (index space)
- Local Dimensions: 64 (work-items per work-group)
  Only 1024/64 = 16 work-groups in total



- Important implication: we will have a lot fewer work-items per work-group (64) and work-groups (16). Why might this matter?

Matrix multiplication: One work item per row of C

```
__kernel void mmul(
   const int Order,
   __global float *A,
   __global float *B,
   __global float *C)
{
   int j, k;
   int i = get_global_id(0);
   float tmp;
   for (j = 0; j < Order; j++) {
        tmp = 0.0f;
        for (k = 0; k < Order;
k++)
             tmp +=
A[i*Order+k]*B[k*Order+j];
        C[i*Order+j] = tmp;
    }
}
```

Mat. Mul. host program (1 row per work-item)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{  // declarations (not shown)
  sz = N * N;
  std::vector<float> h_A(sz);
  std::vector<float> h_B(sz);
  std::vector<float> h_C(sz);

 cl::Buffer d_A, d_B, d_C;

// initialize matrices  and setup
// the problem (not shown)

 cl::Context context(DEVICE);
 cl::Program program(context,
    util::loadProgram("mmulCrow.cl“,
        true));
```

```
cl::CommandQueue queue(context);

auto mmul = cl::make_kernel
        <int, cl::Buffer, cl::Buffer, cl::Buffer>
                        (program, "mmul");

  d_A   = cl::Buffer(context, begin(h_A),
                                end(h_A), true);
  d_B   = cl::Buffer(context, begin(h_B),
                                end(h_B), true);
  d_C   = cl::Buffer(context,
                     CL_MEM_WRITE_ONLY,
                      sizeof(float) * sz);

  mmul(cl::EnqueueArgs( queue,
                        cl::NDRange(N),
                        cl::NdRange(64)),
                        N, d_A,  d_B,  d_C);

  cl::copy(queue, d_C, begin(h_C),
                                end(h_C));

   // Timing and check results (not shown)
}
```

Mat. Mul. host program (1 row per work-item)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{  // declarations (not shown)
  sz = N * N;
  std::vector<float> h_A(sz);
  std::vector<float> h_B(sz);
  std::vector<float> h_C(sz);
```

Changes to host program:

1. 1D ND Range set to number of rows in the C matrix
2. Local Dimension set to 64 (which gives us 16 work-groups which matches the GPU's number of compute units).

```
  util::loadProgram("mmulCrow.cl",
      true));
```

```
cl::CommandQueue queue(context);

auto mmul = cl::make_kernel
        <int, cl::Buffer, cl::Buffer, cl::Buffer>
                        (program, "mmul");

  d_A  = cl::Buffer(context, begin(h_A),
                                  end(h_A), true);
  d_B  = cl::Buffer(context, begin(h_B),
                                  end(h_B), true);
  d_C  = cl::Buffer(context,
                    CL_MEM_WRITE_ONLY,
                     sizeof(float) * sz);

  mmul(cl::EnqueueArgs( queue,
                        cl::NDRange(N),
                        cl::NdRange(64)),
                        N, d_A,  d_B,  d_C);

  cl::copy(queue, d_C, begin(h_C),
                                  end(h_C));

   // Timing and check results (not shown)
}
```

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8

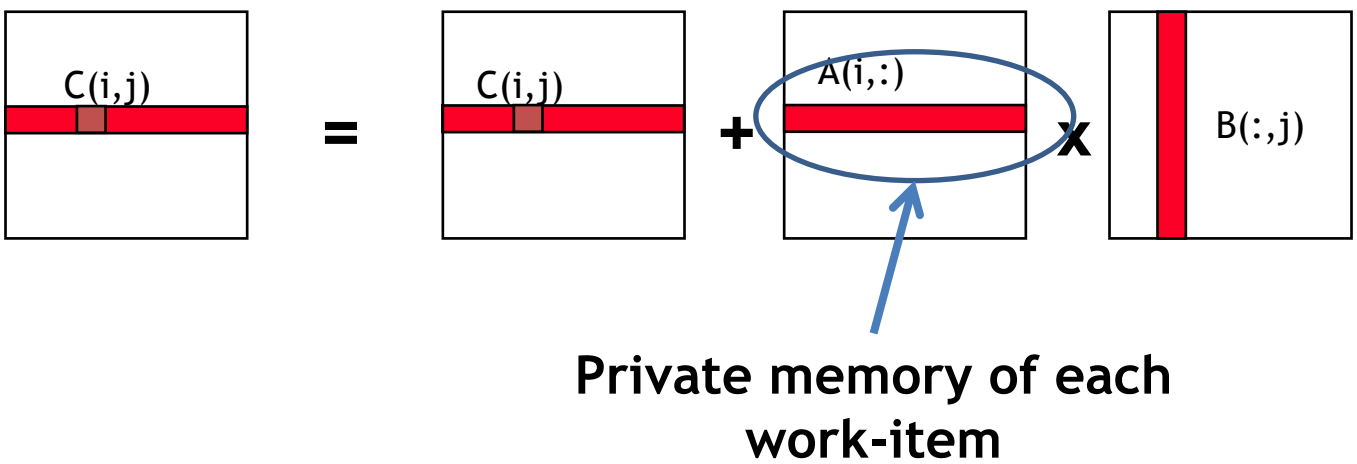
This has started to help.

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

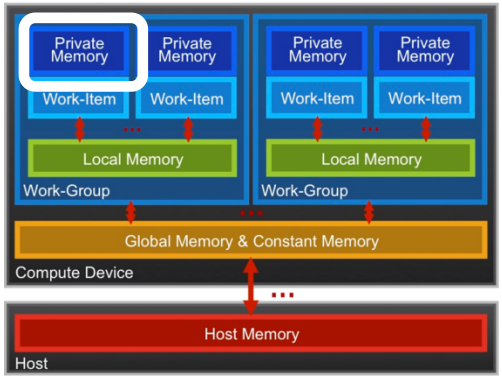
These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Optimizing matrix multiplication

- Notice that, in one row of C, each element reuses the same row of A.
- Let's copy that row of A into private memory of the work-item that's (exclusively) using it to avoid the overhead of loading it from global memory for each C(i,j) computation.



Private Memory



- A work-items private memory:
  - A **very scarce** resource, only a few tens of 32-bit words per Work-Item at most (on a GPU)
  - If you use too much it spills to global memory or reduces the number of Work-Items that can be run at the same time, potentially harming performance*
  - Think of these like registers on the CPU
- How do you create and manage private memory?
  - Declare statically inside your kernel

* Occupancy on a GPU

Matrix multiplication: (Row of A in private memory)

```
__kernel void mmul(
    const int Order,
    __global float *A,
    __global float *B,
    __global float *C)
{
   int j, k;
   int i =
get_global_id(0);
   float tmp;
   float Awrk[1024];
```

```
for (k = 0; k < Order; k++)
   Awrk[k] = A[i*Order+k];

   for (j = 0; j < Order; j++) {
      tmp = 0.0f;
      for (k = 0; k < Order; k++)
   tmp += Awrk[k]*B[k*Order+j];

         C[i*Order+j] = tmp;
    }
}
```

Matrix multiplication: (Row of A in private memory)

```
__kernel void mmul(
    const int Order,
    __global float *A,
    __global float *B,
    __global float *C)
{
   int j, k;
   int i =
get_global_id(0);
   float tmp;
   float Awrk[1024];
```

Setup a work array for A in private memory*

```
for (k = 0; k < Pdim; k++)
    Awrk[k] = A[i*Ndim+k];

   for (j = 0; j < Orde
     tmp = 0.0f;
     for (k = 0; k < Or
   tmp += Awrk[k]*B[k*C

        C[i*Order+j] = tmp;
    }
}
```

Copy a row of A into private memory from global memory before we start with the matrix multiplications.

(*Actually, this is using ***far*** more private memory than we'll have and so Awrk[] will be spilled to global memory)

Mat. Mul. host program (Row of A in private memory)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{  // declarations (not shown)
  sz = N * N;
  std::vector<float> h_A(sz);
  std::vector<float> h_B(sz);
  std::vector<float> h_C(sz);

 cl::Buffer d_A, d_B, d_C;

// initialize matrices  and setup
// the problem (not shown)

 cl::Context context(DEVICE);
 cl::Program program(context,
    util::loadProgram("mmulCrow.cl“,
        true));
```

```
cl::CommandQueue queue(context);

auto mmul = cl::make_kernel
        <int, cl::Buffer, cl::Buffer, cl::Buffer>
                        (program, "mmul");

  d_A   = cl::Buffer(context, begin(h_A),
                                end(h_A), true);
  d_B   = cl::Buffer(context, begin(h_B),
                                end(h_B), true);
  d_C   = cl::Buffer(context,
                    CL_MEM_WRITE_ONLY,
                     sizeof(float) * sz);

  mmul(cl::EnqueueArgs( queue,
                     cl::NDRange(N),
                     cl::NDRange(64)),
                     N, d_A,  d_B,  d_C);

  cl::copy(queue, d_C, begin(h_C),
                                end(h_C));

   // Timing and check results (not shown)
}
```

Host program unchanged from last exercise

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8
C row per work-item, A row private	3,385.8	8,584.3

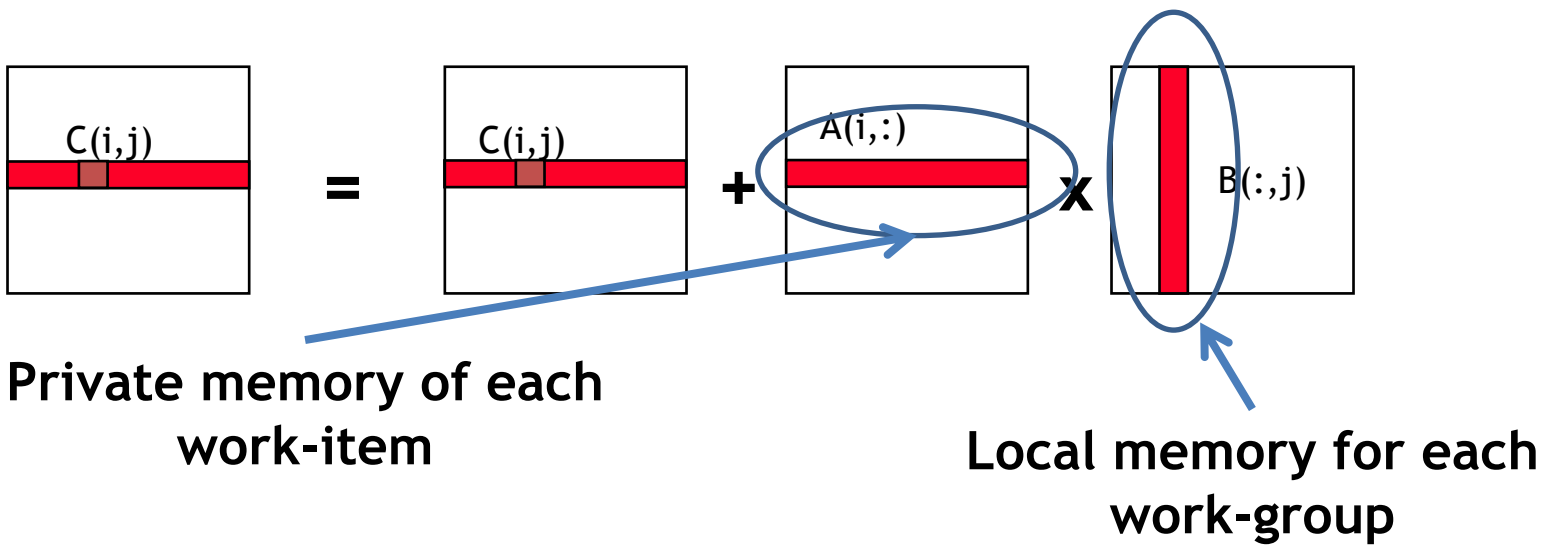
Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

Big impact!

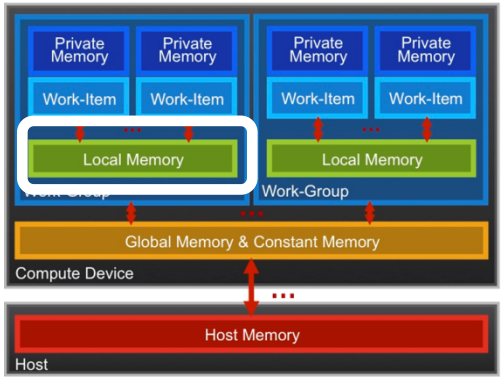
These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Optimizing matrix multiplication

- We already noticed that, in one row of C, each element uses the same row of A
- Each work-item in a work-group also uses the same columns of B
- So let's store the B columns in local memory (which is shared by the work-items in the work-group)



Local Memory



- A work-group's shared memory
  - Typically 10's of KBytes per Compute Unit*
  - Use Local Memory to hold data that can be **reused by all the work-items** in a work-group
  - As multiple Work-Groups may be running on each Compute Unit (CU), only a fraction of the total Local Memory size may be available to each Work-Group
- How do you create and manage local memory?
  - Create and Allocate local memory on the host

```
cl::LocalSpaceArg  localmem = cl::Local(sizeof(float)* N);
```

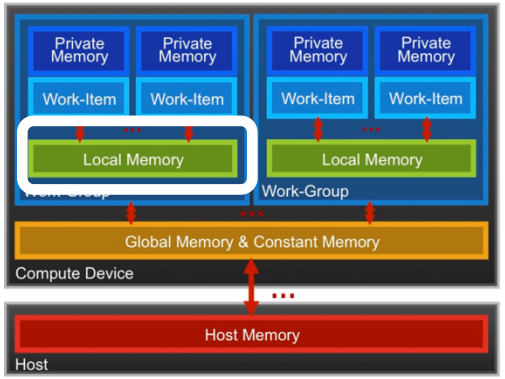
  - Setup the kernel to receive local memory blocks

```
auto foo = cl::make_kernel<int, cl::Buffer,
cl::LocalSpaceArg>(program, "bar");
```

  - Mark kernel arguments that are from local memory as __local
  - Your kernels are responsible for transferring data between Local and Global/Constant memories … there are built-in functions to help (async_work_group_copy(), async_workgroup_strided_copy(), etc)

*Size and performance numbers are approximate and for a high-end discrete GPU, circa 2011

Local Memory performance hints



- Local Memory doesn't always help…
  - CPUs don't have special hardware for it
  - This can mean excessive use of Local Memory might slow down kernels on CPUs
  - GPUs now have effective on-chip caches which can provide much of the benefit of Local Memory but without programmer intervention
  - Access patterns to Local Memory affect performance in a similar way to accessing Global Memory
    - Have to think about things like coalescence & bank conflicts
  - So, your mileage may vary!

Memory Consistency

- OpenCL uses a **relaxed consistency** memory model; i.e.
  - The state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times.
- Within a work-item:
  - Memory has load/store consistency to the work-item's private view of memory, i.e. it sees its own reads and writes correctly
- Within a work-group:
  - Local memory is consistent between work-items at a barrier.
- Global memory is consistent within a work-group at a barrier, **but *not* guaranteed across different work-groups!!**
  - This is a common source of bugs!
- Consistency of memory shared between commands (e.g. kernel invocations) is enforced by synchronization (barriers, events, in-order queue)

Work-Item Synchronization

Ensure correct order of memory operations to local or global memory (with flushes or queuing a memory fence)

- Within a work-group

  `void barrier()`

  – Takes optional flags

    CLK_LOCAL_MEM_FENCE and/or CLK_GLOBAL_MEM_FENCE

  – A work-item that encounters a **barrier**() will wait until ALL work-items in its work-group reach the **barrier**()

  – **Corollary**: If a **barrier**() is inside a branch, then the branch must be taken by either:

    - **ALL** work-items in the work-group, OR
    - **NO** work-item in the work-group

- Across work-groups

  – No guarantees as to where and when a particular work-group will be executed relative to another work-group

  – **Cannot exchange data, or have barrier-like synchronization between two different work-groups! (Critical issue!)**

  – **Only solution: finish the kernel and start another**

Matrix multiplication: B column shared between work-items

```
__kernel void mmul(
        const int Order,
   __global float *A,
   __global float *B,
   __global float *C,
   __local   float *Bwrk)
{
 int j, k;
 int i = get_global_id(0);

 int iloc = get_local_id(0);
 int nloc= get_local_size(0);

 float tmp;
 float Awrk[1024];
```

```
for (k = 0; k < Order; k++)
   Awrk[k] = A[i*Order+k];

 for (j = 0; j < Order; j++) {

   for (k=iloc; k< Order; k+=nloc)
        Bwrk[k] = B[k* Order +j];
   barrier(CLK_LOCAL_MEM_FENCE);

   tmp = 0.0f;
   for (k = 0; k < Order; k++)
       tmp += Awrk[k]*Bwrk[k];

   C[i*Order+j] = tmp;
   barrier(CLK_LOCAL_MEM_FENCE);
 }
}
```

Matrix multiplication: B column shared between work-items

```
__kernel void mmul(
        const int Order,
   __global float *A,
   __global float *B,
   __global float *C,
   __local   float *Bwrk)
{
 int j, k;
 int i = get_global_id(0);

 int iloc = get_local_id(0);
 int nloc= get_local_size(0);

 float tmp;
 float Awrk[1024];

 for (k = 0; k < Order; k++)
    Awrk[k] = A[i*Order+k];

 for (j = 0; j < Order; j++) {

   for (k=iloc; k< Order; k+=nloc)
        Bwrk[k] = B[k* Order +j];
   barrier(CLK_LOCAL_MEM_FENCE);

   tmp = 0.0f;
   for (k = 0; k < Order; k++)
       tmp += Awrk[k]*Bwrk[k];

   C[i*Order+j] = tmp;
   barrier(CLK_LOCAL_MEM_FENCE);
 }
}
```

Pass a work array in local memory to hold a column of B. All the work-items do the copy "in parallel" using a cyclic loop distribution (hence why we need iloc and nloc)

Mat. Mul. host program (Share a column of B within a work-group)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{  // declarations (not shown)
  sz = N * N;
  std::vector<float> h_A(sz);
  std::vector<float> h_B(sz);
  std::vector<float> h_C(sz);

 cl::Buffer d_A, d_B, d_C;

// initialize matrices  and setup
// the problem (not shown)

 cl::Context context(DEVICE);
 cl::Program program(context,
    util::loadProgram("mmulCrow.cl“,
        true));
```

```
cl::CommandQueue queue(context);

auto mmul = cl::make_kernel
       <int, cl::Buffer, cl::Buffer, cl::Buffer,
             cl::LocalSpaceArg > (program, "mmul");

  d_A   = cl::Buffer(context, begin(h_A), end(h_A),true);
  d_B   = cl::Buffer(context, begin(h_B), end(h_B),true);
  d_C   = cl::Buffer(context,
            CL_MEM_WRITE_ONLY, sizeof(float) * sz);

  cl::LocalSpaceArg Bwrk =
                 cl::Local(sizeof(float) * Pdim);

  mmul(cl::EnqueueArgs( queue,
              cl::NDRange(N),  cl::NDRange(64)),
               N, d_A,  d_B,  d_C, Bwrk);

  cl::copy(queue, d_C, begin(h_C), end(h_C));

   // Timing and check results (not shown)
}
```

Mat. Mul. host program (Share a column of B within a work-group)

```
#define DEVICE CL_DEVICE_TYPE_DEFAULT
int main(void)
{  // declarations (not shown)
  sz = N * N;
  std::vector<float> h_A(sz);

 c

/
/

 cl::Context context(DEVICE);
 cl::Program program(context,
    util::loadProgram("mmulCrow.cl“,
        true));
```

Change host program to pass local memory to kernels.

- Add an arg of type LocalSpaceArg is needed.
- Allocate the size of local memory
- Update argument list in kernel functor

```
cl::CommandQueue queue(context);

auto mmul = cl::make_kernel
      <int, cl::Buffer, cl::Buffer, cl::Buffer,
            cl::LocalSpaceArg > (program, "mmul");

  d_A   = cl::Buffer(context, begin(h_A), end(h_A),true);
  d_B   = cl::Buffer(context, begin(h_B), end(h_B),true);
  d_C   = cl::Buffer(context,
            CL_MEM_WRITE_ONLY, sizeof(float) * sz);

  cl::LocalSpaceArg Bwrk =
                cl::Local(sizeof(float) * Pdim);

  mmul(cl::EnqueueArgs( queue,
              cl::NDRange(N),  cl::NDRange(64)),
               N, d_A,  d_B,  d_C, Bwrk);

  cl::copy(queue, d_C, begin(h_C), end(h_C));

   // Timing and check results (not shown)
}
```

Matrix multiplication performance

- Matrices are stored in global memory.

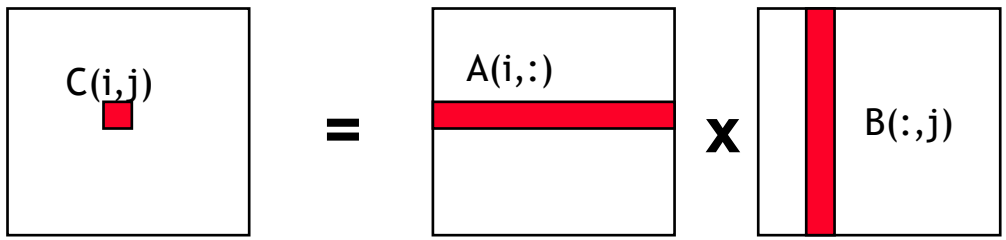
Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8
C row per work-item, A row private	3,385.8	8,584.3
C row per work-item, A private, B local	10,047.5	8,181.9

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

The CuBLAS SGEMM provides an effective measure of peak achievable performance on the GPU. CuBLAS performance = 283366.4 MFLOPS

Matrix multiplication example:
Naïve solution, one dot product per element of C

- Multiplication of two dense matrices.



Dot product of a row of A and a column of B for each element of C

- To make this fast, you need to break the problem down into chunks that do lots of work for sub problems that fit in fast memory (OpenCL local memory).

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
        for (k = 0; k < N; k++) {
          C[i*N+j] += A[i*N+k] * B[k*N+j];
        }
      }
    }
}
```

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
    int i, j, k;
    for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
          C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Let's get rid of all those ugly brackets

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (i = ib*NB; i < (ib+1)*NB; i++)
     for (jb = 0; jb < NB; jb++)
       for (j = jb*NB; j < (jb+1)*NB; j++)
         for (kb = 0; kb < NB; kb++)
           for (k = kb*NB; k < (kb+1)*NB; k++)
             C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Break each loop into chunks with a size chosen to match the size of your fast memory

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

Rearrange loop nest to move loops over blocks “out” and leave loops over a single block together

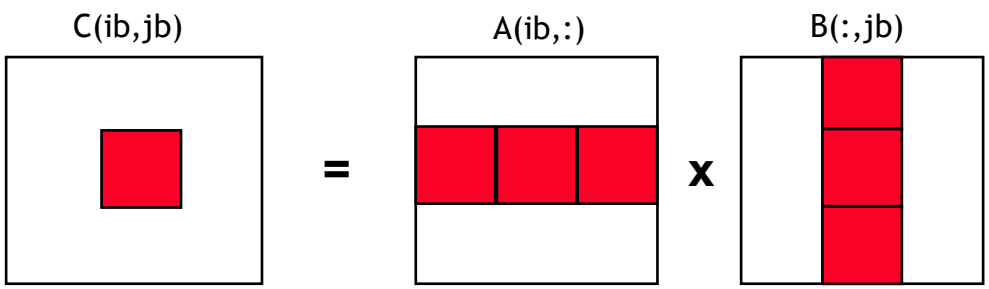
Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 float tmp;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)

 for (i = ib*NB; i < (ib+1)*NB; i++)
   for (j = jb*NB; j < (jb+1)*NB; j++)
     for (k = kb*NB; k < (kb+1)*NB; k++)
       C[i*N+j] += A[i*N+k] * B[k*N+j];
}
```

This is just a local matrix multiplication of a single block

Matrix multiplication: sequential code

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)
       sgemm(C, A, B, …)    // C_ib,jb = A_ib,kb * B_kb,jb
}
```

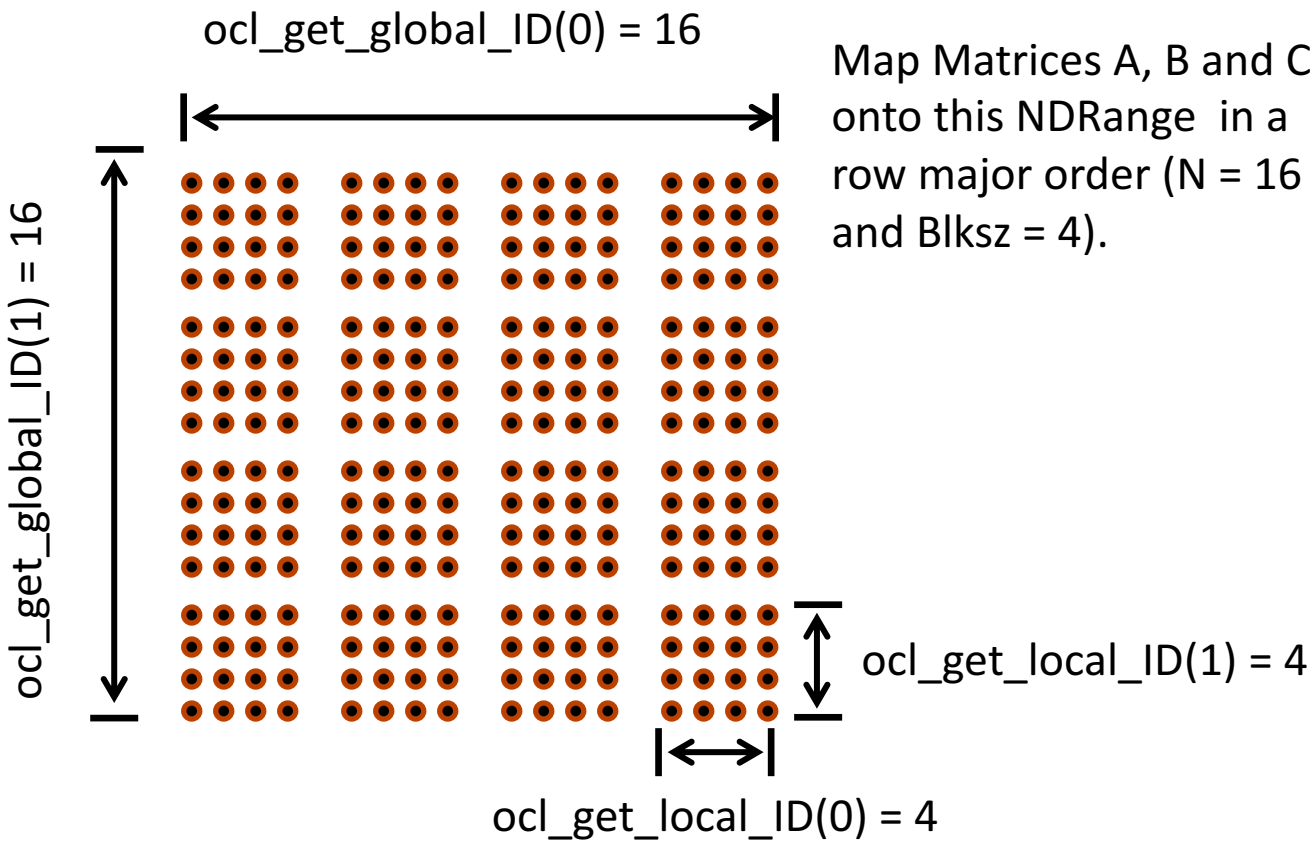


Note: sgemm is the name of the level three BLAS routine to multiply two matrices

Mapping into A, B, and C from each work item

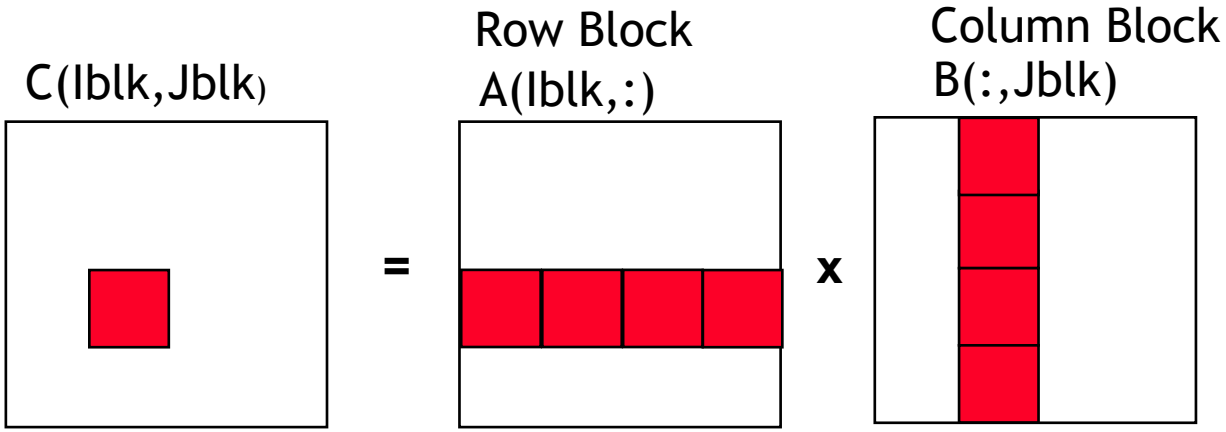
Understanding index offsets in the blocked matrix multiplication program.

16 x 16 NDRange with workgroups of size 4x4



Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).

Mapping into A, B, and C from each work item

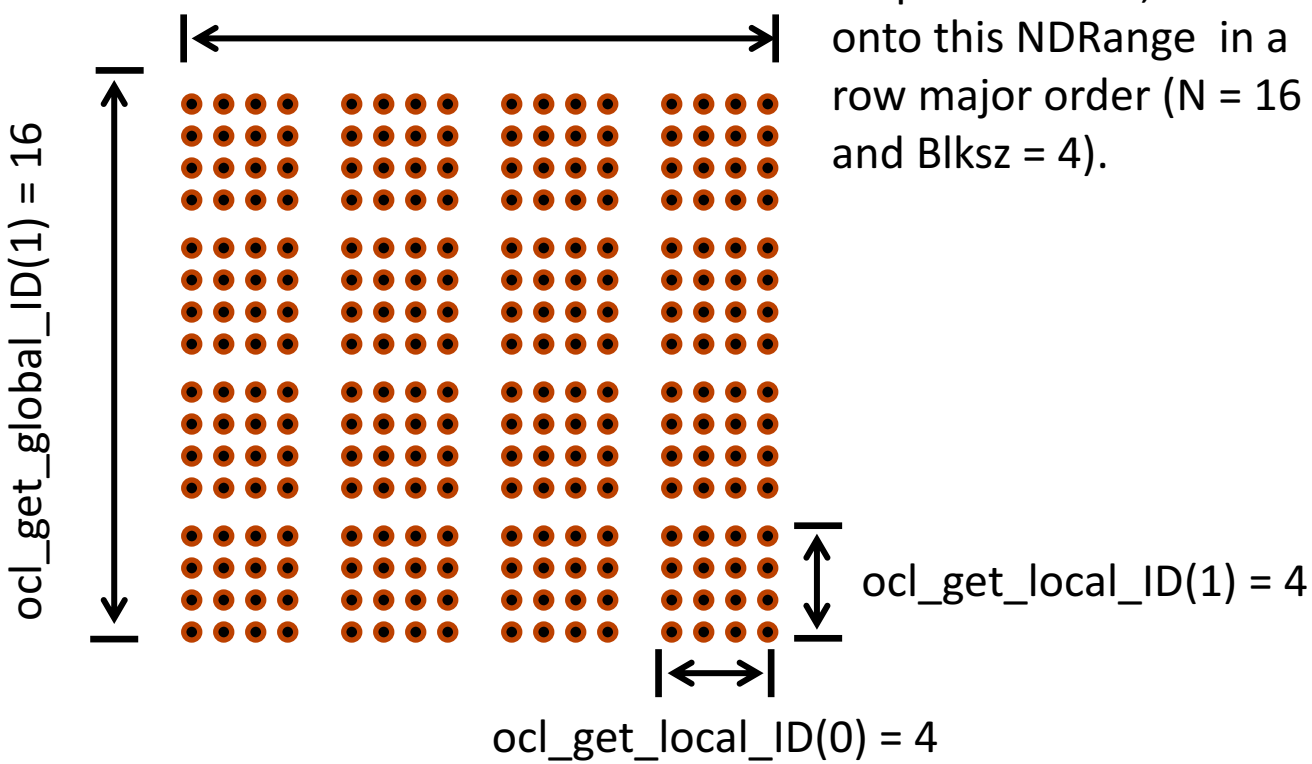


Understanding index offsets in the blocked matrix multiplication program.

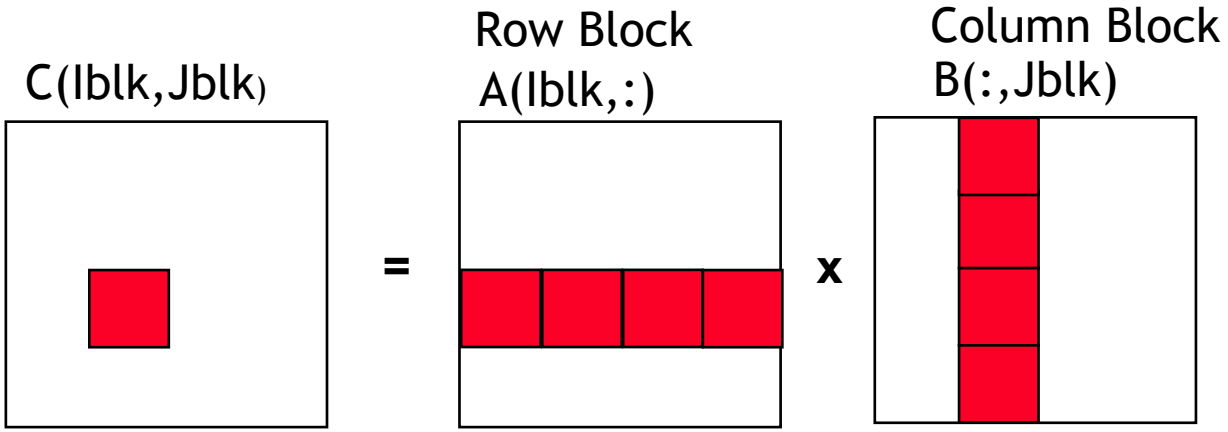
16 x 16 NDRange with workgroups of size 4x4

ocl_get_global_ID(0) = 16

Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).



Mapping into A, B, and C from each work item



Understanding index offsets in the blocked matrix multiplication program.

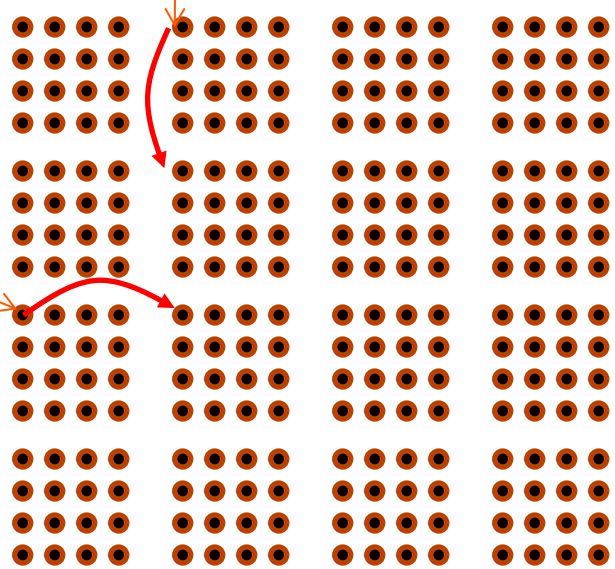
16 x 16 NDRange with workgroups of size 4x4

Consider indices for computation of the block C(Iblk=2, Jblk=1)

Bbase = Jblk*blksz = 1*4

Map Matrices A, B and C onto this NDRange in a row major order (N = 16 and Blksz = 4).

Abase = Iblk*N*blksz
= 1 * 16 * 4



Subsequent B blocks by shifting index by
Binc = blksz * N
= 4 * 16 = 64

Subsequent A blocks by shifting index by
Ainc = blksz = 4

Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
            const unsigned int N,
            __global float* A,
            __global float* B,
            __global float* C,
            __local  float* Awrk,
            __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;
```

```
  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
      //Each work-item loads a single element of the two
      //blocks which are shared with the entire work-group

      Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
      Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      #pragma unroll
      for(kloc=0; kloc<blksz; kloc++)
  Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      Abase += Ainc;    Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
            const unsigned int N,
            __global float* A,
            __global float* B,
            __global float* C,
            __local  float* Awrk,
            __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;
```

Load A and B blocks, wait for all work-items to finish

```
  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
      //Each work-item loads a single element of the two
      //blocks which are shared with the entire work-group

      Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
      Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      #pragma unroll
      for(kloc=0; kloc<blksz; kloc++)
  Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      Abase += Ainc;    Bbase += Binc;
  }
  C[j*N+i] = Ctmp;
}
```

Wait for everyone to finish before going to next iteration of Kblk loop.

Matrix multiplication … Portable Performance

- Single Precision matrix multiplication (order 1000 matrices)

	CPU	Xeon Phi	Core i7, HD Graphics	NVIDIA Tesla
Sequential C (compiled /O3)	224.4		1221.5	
C(i,j) per work-item, all global	841.5	13591		3721
C row per work-item, all global	869.1	4418		4196
C row per work-item, A row private	1038.4	24403		8584
C row per work-item, A private, B local	3984.2	5041		8182
Block oriented approach using local (blksz=16)	12271.3	74051 (126322*)	38348 (53687*)	119305
Block oriented approach using local (blksz=32)	16268.8			

Xeon Phi SE10P, CL_CONFIG_MIC_DEVICE_2MB_POOL_INIT_SIZE_MB = 4 MB

* The comp was run twice and only the second time is reported (hides cost of memory movement.

Intel® Core™ i5-2520M CPU @2.5 GHz (dual core) Windows 7 64 bit OS, Intel compiler 64 bit version 13.1.1.171, OpenCL SDK 2013, MKL 11.0 update 3.

47 @ 2.3 GHz which has an Intel HD Graphics 5200 w/ high speed memory. ICC 2013 sp1 update 2.

Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Matrix multiplication performance

- Matrices are stored in global memory.

Case	MFLOPS	
	CPU	GPU
Sequential C (not OpenCL)	887.2	N/A
C(i,j) per work-item, all global	3,926.1	3,720.9
C row per work-item, all global	3,379.5	4,195.8
C row per work-item, A row private	3,385.8	8,584.3
C row per work-item, A private, B local	10,047.5	8,181.9

Device is Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs
Device is Intel® Xeon® CPU, E5649 @ 2.53GHz

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Matrix multiplication performance (CPU)

- Matrices are stored in global memory.

Case	MFLOPS CPU
Sequential C (not OpenCL, compiled /O3)	224.4
C(i,j) per work-item, all global	841.5
C row per work-item, all global	869.1
C row per work-item, A row private	1038.4
C row per work-item, A private, B local	3984.2
Block oriented approach using local (blksz=8)	7482.5
Block oriented approach using local (blksz=16)	12271.3
Block oriented approach using local (blksz=32)	16268.8
Intel MKL SGEMM	63780.6

Device is Intel® Core™ i5-2520M CPU @2.5 GHz (dual core) Windows 7 64 bit OS, Intel compiler 64 bit version 13.1.1.171, OpenCL SDK 2013, MKL 11.0 update 3.

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

Outline

- OpenCL: overview and core models
- Host programs
- Kernel programs
- Optimizing OpenCL kernels
  - Memory coelescence
  - Divergent control flows
  - Occupancy
  - Other Optimizations
- Working with the OpenCL Memory Hierarchy
- Resources supporting OpenCL

OpenCL 2.0

- OpenCL 2.0 was ratified in Nov'13
- Brings several important new features:
  - Shared Virtual Memory
  - Nested parallelism
  - Built-in work-group reductions
  - Generic address space
  - Pipes
  - C11 atomics
- Specification and headers available here
- Production drivers now available from Intel and AMD, with more expected to follow

- Standard Portable Intermediate Representation
- Defines an IR for OpenCL programs
- Means that developers can ship portable binaries instead of their OpenCL source
- Also intended to be a target for other languages/programming models (C++ AMP, SYCL, OpenACC, DSLs)
- SPIR 1.2 & SPIR 2.0 ratified, SPIR-V provisional available now
- Implementations available from Intel and AMD, with more on the way

- Single source C++ abstraction layer for OpenCL
- Goal is to enable the creation of C++ libraries and frameworks that utilize OpenCL
- Can utilize SPIR to target OpenCL platform
- Supports ‘host-fallback’ (CPU) when no OpenCL devices available
- Provisional specification released Mar’14
- Codeplay and AMD working on implementations

Libraries

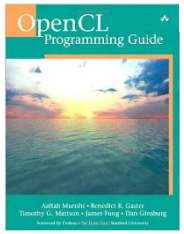
- clFFT/clBLAS / clRNG (all on github)
- Arrayfire (open source soon)
- Boost compute with VexCL
- ViennaCL (PETSc), PARALUTION
- Lots more - see the Khronos OpenCL pages:

https://www.khronos.org/opencl/resources

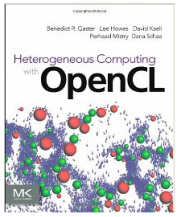
Resources: https://www.khronos.org/opencl/

The OpenCL specification
Surprisingly approachable for a spec!
https://www.khronos.org/registry/cl/

OpenCL reference card
Useful to have on your desk(top)
Available on the same page as the spec.



OpenCL Programming Guide:
Aaftab Munshi, Benedict Gaster, Timothy G. Mattson and James Fung, 2011



Heterogeneous Computing with OpenCL
Benedict Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry and Dana Schaa, 2011

OpenCL Tutorials

http://handsonopencl.github.io

- One of the most popular OpenCL training courses on the web
- Completely open source (creative commons attribution CC BY license)
- Downloaded over 4,200 times so far!
- Lots of training material, examples and solutions, source code etc
- Works on Linux, Windows, OSX etc.

Other useful resources

- Lots of OpenCL examples in the SDKs from the vendors:
  - AMD, Intel, Nvidia, …
- The SHOC OpenCL/CUDA benchmark suite (available as source code):
  - https://github.com/vetter/shoc/wiki
- The GPU-STREAM memory bandwidth benchmark:
  - https://github.com/UoB-HPC/GPU-STREAM

Other useful resources

- IWOCL webpage & newsletter:
  - http://www.iwocl.org
  - http://www.iwocl.org/signup-for-updates/

- IWOCL annual conference
  - Spring each year
  - In Vienna, April 19-21 2016!

IWOCL
INTERNATIONAL WORKSHOP ON OPENCL

Conclusion

- OpenCL
  - Widespread industrial support
  - Defines a platform-API/framework for heterogeneous parallel computing, not just GPGPU or CPU-offload programming
  - Has the potential to deliver portably performant code; but it has to be used correctly

Notes for next time

- Add jac_solv_ocl_basic to the makefile we give the students.
- Add a make defs directory. Include the clang case.
- Get rid of OpenMP. Then the code would work with Clang or g++
- Shorten the names of the jac_solv_ocl cases … it's a pain typing them all the time.
- Add more comments to the cl template so it is more clear what we want the students to do.
- I need to merge the set of slides so the wordy slide 50 is developed one by one with the pictures that follow. Talking through slide 50 than showing the pictures just doesn't work.
- Modify mm_utils.c to include the colmaj generator. I like that better than providing the host program with the new generator.

- Yes, they were all OpenCL times (double precision). The CPU is a dual-socket Intel(R) Xeon(R) CPU E5-2687W (16 cores total, with hyper-threading enabled). I've attached the output of a clinfo run on this machine. Your jac_solv_parfor (compiled with icc) achieves this on the CPU:

- 25.3 seconds (32 threads)
- 19.0 seconds (OMP_NUM_THREADS=16, to avoid hyper-threading)

- The serial code takes 83 seconds.

- Running the OpenMP version natively on the Xeon Phi gives a very impressive time of 4.8 seconds.
-
- > As Tom says, most GPUs will need a large matrix to really get going. Here's the timings I get with Ndim=4096 when running on four different devices (NVIDIA GPU, AMD GPU, Xeon Phi and Xeon CPU).
- >
- > ----------
- > | | K40 | 290X | Phi | Xeon |
- > |--------|---------|---------|---------|---------|
- > | basic | 35.0 | 198.2 | 245.2 | 23.6 |
- > | colmaj | 14.1 | 15.3 | 35.8 | 71.5 |
- > | nobr | 13.3 | 15.6 | 16.6 | 38.8 |
- > | wg | 13.2 | 15.1 | 15.0 | 36.8 |
- > | best | 6.2 | 6.7 | 13.3 | 32.1 |
- > ----------
- >